UPDATE .... KYAN!

Kyan Software
1850 Union Street #183
San Francisco, CA 94123

Volume 1 Number 1
(c) 1985 Kyan Software
November/December Issue

Apple Edition

What's New

Welcome to "UPDATE .... KYAN"

Welcome to the first issue of "UPDATE .... KYAN", the bimonthly newsletter published by Kyan Software for registered owners of Kyan Software products. The purpose of "UPDATE .... KYAN" is to provide you with new information about Pascal programming and to keep you posted on product updates and new product releases. We encourage you to subscribe.

"UPDATE .... KYAN" is organized into five sections.

WHAT'S NEW .... a section containing news about Kyan Software, product revisions and updates, and new product releases.

KYAN PASCAL UPDATE .... a section containing news and information about Kyan Pascal. This section describes how the software works and the rationale behind the design. It also discusses undocumented features in the software and other subjects not fully covered in the manual.

PASCAL PROGRAMMING .... a section containing programming tips, new Pascal routines, bug reports and workarounds, and other information which will be useful to you when programming in Pascal. Contributions from Kyan Pascal users are encouraged.

ASSEMBLY LANGUAGE PROGRAMMING .... a section containing programming tips and new assembly language routines which can be used in your programs. Contributions to this corner are also solicited from Kyan Pascal users.

LETTERS to the Editor .... a section containing letters and questions received from users of Kyan Pascal. Many of these letters express opinions or contain questions which may be of interest to you. We encourage you to tell us what you like or don't like about Kyan Software products. Write us about what new products you would like to see from Kyan Software.

We are quite excited about "UPDATE .... KYAN". We believe it will become a valuable resource to our users and another good reason to purchase software products from Kyan Software.

A Little Background on Kyan Software

Kyan Software was formed in early 1984 to develop professional quality, low cost, programming language implementations for microcomputers. Kyan's first product, a p-code compiler for the Commodore 64, was introduced in August 1984 and was well received by schools and students as a tool for learning the Pascal language.

Kyan's second product was a machine code Pascal compiler which would run on any computer with a 6502 microprocessor. This new compiler was introduced in Spring of 1985 and became the first full implementation of Jensen/Wirth Pascal for the Apple II, Atari and Commodore family of computers.

Kyan Software is still a small company with only a handful of full and part time employees. We are continuing to work on upgrades to our Pascal compilers and are expanding the product line with toolkits for Pascal programmers. We are also working on several new programming languages which will be introduced in 1986.

Kyan Software is thankful for the support of our customers. To show our appreciation, Kyan Software adhers to a policy of offering revisions and upgrades to registered owners at little or no charge. We want to be sure that each of you is working with the latest release of the software. We also offer you advance notice of new products and special discounts on the new software. We thank you for supporting Kyan Software and look forward to serving you in the future.

Kyan Pascal ... Version 1.2

We are now shipping Version 1.2 of Kyan Pascal for the Apple. This revision corrects a couple of bugs which were uncovered in Versions 1.0 and 1.1 and contains a faster graphics algorithm. If the slow graphics annoy you or you keep bumping into what you think is a bug in the software, then return your Kyan Pascal source disk for a free upgrade. If you haven't noticed any problems or you can hold out for a while longer, we ask that you wait for a major upgrade which will be announced in the next issue of **"UPDATE .... KYAN"**.

KYAN PASCAL ..... A Product in Evolution

A programming language is different from other types of software. Unlike a word processing or spreadsheet package, it is extremely difficult to define all of the possible uses of the software. As an analogy, consider a spoken language such as English or French; how many different ways are there to use the language syntax? You can write a poem, a letter, a rock video, or the great (or not so great) American novel. Will the language support all of these applications? Will the typewriter you are using have all the symbols and characters you need? Will the reader of your creation know what all the symbols, characters, and words mean?

We are confronted with similar questions when implementing a programming language. Jensen and Wirth solved most of the Pascal syntax problems. But, Kyan must deal with the problems of accurately interpreting this syntax and correctly compiling a listing which is meaningful to the computer. We constantly face the question, "what program construction (legal or illegal) can cause a crash during compilation or runtime?"

During the development and beta testing of each new product, we subject it to a battery of test programs to make sure it works properly under as many conditions as possible. When we release the product to the market, we have a high level of confidence that it will perform in a satisfactory manner.

However, users inevitably write programs which uncover a bug which we missed. When this occurs, the customer calls our tech support group and points out the problem (sometimes in very graphic terms!). In 99 cases out of 100, we are able to quickly correct the problem and send the customer a new disk. This fix is then added to the list of changes which will be released in the next general REVISION of the compiler (i.e., version 1.1 to 1.2).

Over a period of time, the bugs we find become far more subtle -- 99% of the users would never encounter them. But, since we want to ship the best possible product, we strive to document and fix every bug identified.

Then, just when the product is "perfect", the engineers come up with some new enhancements to the product (i.e., "let's increase the size of the symbol table and add some new extensions!"). We then go through another development cycle and release a product UPGRADE (i.e., Version 1.3 to 2.0). And, the whole process begins again.

So, a programming language product like **Kyan Pascal** is never done ... it is constantly evolving to a better and more refined state. We can never say with absolute certainty that it is "bug-free". However, what we can say is that when you buy a product from Kyan Software, you will receive the highest quality possible, good technical support, and periodic revisions and upgrades at the lowest possible cost.

Kyan Pascal Update

We are frequently asked questions about how Kyan Pascal is designed and how it operates. The following notes should give you a better understanding of how Kyan Pascal works.

The Kyan Pascal Compiler/Assembler

Kyan Pascal is a comprised of a Pascal compiler which produces assembly language output and an assembler which produces an executable file. When you compile a Pascal source code file, the compiler produces assembly source code that is optimized for runtime speed. The assembler takes this source code and reads it twice; on the first pass, it builds the assembler symbol table; on the second pass, it produces 6502 machine code. This two pass approach allows all forward references to be resolved and results in generation of the fastest and most efficient machine code possible for the 6502 processor.

Runtime Benchmarks and Arithmetic Precision

Kyan Pascal produces code that runs about twice as fast on a 6502 microprocessor as the best selling Pascal does on a Z80 (assuming equal CPU clock rates). The benchmark used for this comparison is the Sieve Algorithm and the time required to generate the first 1,399 prime numbers (execution time: 12 seconds).

The arithmetic used in Kyan Pascal is either 16 bit integer or 13 decimal digit BCD. Kyan Pascal uses BCD real numbers to eliminate round-off errors of binary representations. (Who wants to have a result of a simple division displayed as 2.99999999 instead of 3.0?)

Calculation speed is proportional to the square of the precision of the real number. In floating point benchmarks, Kyan Pascal produces code that runs at approximately the same speed as compilers with 7 to 9 digit precision. For equivalent precision, Kyan Pascal is therefore running 2 to 4 times faster.

Source Code vs. Object Code Linking

In Kyan Pascal you can link program modules together by "including" Pascal or assembly language source files into the main program. An object module linker is not used. We decided not to use an object module linker because it requires two passes of the object modules to produce an executable file. The time required for these two passes is more than that required for Kyan Pascal to recompile the sources. Object module linking has the additional disadvantage of producing non-standard effects on modules of the Pascal program (i.e., no scope rules, no parameter checking, and no mechanism for assigning a lexical level to variables).

Notes about the Kyan Pascal/ProDOS Interface

When you purchase Kyan Pascal, the ProDOS operating system and the ProDOS FILER are included on the diskette. These are the only files you need to develop and run programs using Kyan Pascal. Due to space constraints on the diskette, Kyan Pascal does <u>not</u> contain APPLESOFT BASIC, DOS 3.3 <> ProDOS conversion utilities, or the other utilities found on a complete ProDOS source disk.

Kyan Pascal does not have its own operating system or environment. It operates entirely within ProDOS. When you exit from the editor, compiler, or your own program, you will see the ProDOS prompt (">"); it is asking for the pathname of the next program to be run.

The object module produced by Kyan Pascal is a SYSTEM file. This means you can call and run your programs anytime you see the ProDOS prompt.

Programmers should be careful not to confuse ProDOS file management commands with those of Apple BASIC. File management in ProDOS is accomplished using the ProDOS FILER. When you are programming in Apple BASIC, you have similar capabilities but these are BASIC commands and not ProDOS commands.

BASIC programs (including assembler programs that are in BLOAD format) need to be run from the BASIC environment. This is in contrast to Pascal programs which are run from ProDOS.

Any Pascal program with a ".SYSTEM" extension to the filename will execute immediately after boot-up. The Kyan copyright notice, HELLO.SYSTEM, is just such a program. (Note: Please refer to the PASCAL PROGRAMMING Section for more information about how to set-up an auto-boot program).

Pack and Unpack

PACK and UNPACK are standard procedures in Pascal. Kyan Pascal automatically packs all structures at the byte level. The only variable type which is not fully packed is booleans. Because of the poor bit handling of the 6502 microprocessor, Kyan Pascal does not support the packing of booleans. Our packed and unpacked structures are identical.

Commonly asked questions about Kyan Pascal's Runtime Library

What is the Runtime Library? The Runtime Library is a software module which contains the general purpose routines used in Pascal programs. Library routines include input/output functions, floating point package, transcendental functions, and set routines. The Runtime Library conserves space on the disk. Rather than requiring a copy of every Pascal routine for each program, the Runtime Library allows one copy of the routines to be shared by many programs. Since the Runtime Library is approximately 10K in size, you can see how much disk space you gain if you want to put 3 or 4 programs on the same disk.

Do I need a copy of the Library on each disk? Yes. The Library must be copied onto each disk so that your programs can be run independently of the Kyan Pascal program disk.

Can I just copy parts of the Runtime Library onto the disk? No. The Pascal Library is a single binary file and cannot be broken up or partially loaded.

If I sell my programs with a copy of the Library on the disk, do I need to have a license and pay royalties? No. Kyan Software does not believe you should have to pay us for software you have developed. You are free to use the Kyan Runtime Library with you software provided that you acknowledge Kyan Software's copyright.

How do I acknowledge Kyan's copyright? Simple. Just add the following notice to the diskette label -- "Pascal Library (c) Kyan Software 1985" -- and repeat this notice in the manual along with your own copyright notice.

Pascal Programming

This section of the newsletter is reserved for Pascal programming tips and new routines developed by the Kyan technical staff and by Kyan Pascal users. If you have developed Pascal routines which you think might be of interest to other users, send them to us. We will select the best submissions and publish them with your name.

How to set up programs to boot automatically.

When you boot a disk on your Apple II, the ProDOS operating system directs the computer to seek out and load the first "System" file listed in the directory. "System" files are easy to identify because they have a filename followed by ".SYSTEM".

Your program will automatically load and run if the program name ends in ".SYSTEM". First, format the disk. Then, copy the ProDOS operating system and Pascal Runtime Library onto the disk. Next, rename your program so that the filename ends in ".SYSTEM". Finally, make sure that your program is the first "System" file listed in the directory.

For example, suppose you want your program "Apple" to load and run automatically. First, copy ProDOS and the Runtime Library onto the disk. Then, change the filename of your compiled program from "Apple.O" to "Apple.System" using the file Rename command in the ProDOS FILER. Finally, check to make sure that "Apple.System" is the first System file listed in the directory. Now, whenever this disk is booted, the program "Apple" will automatically load and run.

How to raise a number to a power in Standard Pascal

Standard Pascal has no exponentiation operator. You can work around this limitation in the language by including the following routine in your pascal program.

```
function power(x,n:real):real;
begin
power:=exp(n*ln(x))
end;
```

Page Zero Memory

The page zero memory up to $F is used for pointers to the stack and the heap. Locations $10 to $1F are used as temporaries for the Pascal Runtime Library routines. They are also available to assembler users for use as temporary variables. The remainder of page 0 is used for ProDOS temporaries and an evaluation stack.

The evaluation stack is used to evaluate expressions. The results are moved to the variable stack. The X register of the 6502 is used as the evaluation stack pointer.

The 6502 Stack

The 6502 stack is used to save the return linkage of all subroutines. Procedures and functions can be nested 127 levels deep before the stack overflows.

The Runtime Library

The Kyan Runtime Library resides from $800 to $1FFF and $B000 to $BEFF. The Library is loaded immediately when program control is passed to the Pascal program. The program Library must reside in the same directory as the program object code.

The Kyan Object Code

The compiled code is loaded starting from location $2000 unless relocated upward by an ORG command to make room for the high resolution graphics between $2000 and $3FFF.

The Heap

The Heap starts at the next location above the loaded program and grows toward high memory. The Heap management looks for the first available space of adequate size for dynamic variable storage with the NEW procedure and frees up the used space with the DISPOSE procedure.

The Stack

The Stack starts at $B000 and grows toward low memory. Global variables are referenced from the start of the Stack. Local variables are referenced from the Local pointer. The Stack structure supports access of variables that are at intermediate Lexical levels (neither Local or Global).

ProDOS Error Messages

The following ProDOS error messages supplement those found in the back of your Kyan Pascal manual.

File Types

$00	Typeless file
$04	ASCII text file
$06	General binary file
$0F	Directory file
$C0-EF	ProDOS reserved
$F0	ProDOS added command line
$F1-F8	ProDOS user defined files 1-8
$F9	ProDOS reserved
$FA	Integer BASIC program file
$FB	Integer BASIC variable file
$FC	Applesoft program file
$FD	Applesoft variables file
$FE	Relocatable code file (EDASM)
$FF	ProDOS system file

ProDOS (MLI) Error Codes

$00	No error
$01	Bad system call number
$04	Bad system call parameter count
$25	Interrupt table full
$27	I/O error
$28	No device connected
$2B	Disk write protected
$2E	Disk switched
$40	Invalid pathname
$42	Max number of files open
$43	Invalid reference number
$44	Directory not found
$45	Volume not found
$46	File not found
$47	Duplicate filename
$48	Volume full
$49	Volume directory full
$4A	Incompatible file format, also a ProDOS Directory
$4B	Unsupported storage type
$4C	End of file encountered
$4D	Position out of range
$4E	File access error, also locked file
$50	File is open
$51	Directory structure damaged
$52	Not a ProDOS volume
$53	Invalid system call parameter
$55	Volume Control Block table full
$56	Bad buffer address
$57	Duplicate volume
$5A	File structure damaged

Assembly Language Programming

The Kyan Pascal manual is rather sparse in its treatment of assembly language programming. The following sections provide you with more information about this very powerful feature of Kyan Pascal. (Note: If you want to learn more about assembly language programming, please refer to the Letters Section of the Newsletter for a list of excellent technical reference manuals).

Variable Names in Assembly Language

In assembly language, the Pascal variable names are not available because the assembler has no scope rules like Pascal. Variable offsets must be computed by hand. Consider the following function:

```
function test(a:integer):integer;
var b:integer;
begin
end;
```

The stack frame for this function looks like this:

```
sp      [       ]   lexical level
        [       ]   lsb of "local"
        [       ]   msb of "local"
        [       ]   lsb of "b"
        [       ]   msb of "b"
        [       ]   lsb of "test"
        [       ]   msb of "test"
        [       ]   lsb of "a"
        [       ]   msb of "a"
local
```

The variables are stacked in the reverse order in which they are declared. Integers and enumerated types are allocated 2 bytes; characters and Booleans are allocated 1 byte; and, reals are allocated 8 bytes. Zero page locations "T" through "T+$E" are available as temporary workspace.

With this in mind, we can implement **"peek"** and **"poke"** as follows:

```
procedure poke(loc,val:integer);
begin
#A
        LDY     #5      ;LOC
        LDA     (SP),Y
        STA     T
        INY
        LDA     (SP),Y
        STA     T+1
;
        LDY     #3      ;VAL
        LDA     (SP),Y
;
        LDY     #0      ;POKE
        STA     (T),Y
#
end;
```

```
function peek(loc:integer):integer;
begin
#A
        LDY     #5      ;LOC
        LDA     (SP),Y
        STA     T
        INY
        LDA     (SP),Y
        STA     T+1
;
        LDY     #0      ;PEEK
        LDA     (T),Y
        LDY     #3
        STA     (SP),Y
        INY
        LDA     #0
        STA     (SP),Y
#
end;
```

Assembly Language Interface for Functions

Functions can be used to return a value to an expression directly instead of using a procedure and modifying the actual parameters passed by name (address). The value assigned by the assembler routine to the function name is returned to the expression where the function is called. The function name is located after the passed parameters on the stack and before the local variables. Access to the passed parameters is the same as that for procedures. In the example below, the value of an integer parameter "B" is picked up and a boolean function value is returned.

```
Function XYZ(A,B,C: Integer): Boolean;
Var X: integer;
begin
#A
 .
 .
 LDY #8; OFFSET TO VARIABLE B
 LDA (SP),Y; PUT LSB OF B IN ACC
 STA ....
 INY
 LDA (SP),Y; PUT MSB OF B IN ACC
 STA ....
 .
 .
 .
 .
 LDA ....
 LDY #5; OFFSET TO FUNCTION VARIABLE, XYZ (BOOLEAN)
 STA (SP),Y; SAVE ACC IN XYZ
#
end;
```

In the above example, the offset from the stack pointer to the function name is calculated as follows:

```
  3 bytes offset from (SP) to first local variable
+2  bytes size of local variable, X
  5 total offset
```

The offset to the variable B is:

```
  3 bytes offset from (SP) to first local variable
+2  bytes size of local variable, X
+1  byte for the boolean, XYZ
+2  bytes for the integer, C
  8 total offset
```

Assembly Language Interface for Parameters passed by Name (address)

When parameters are passed by name, the variable's address is passed to the function or procedure rather than the value. If the value of the variable is altered by the called routine, the new value will also be visible in the calling procedure. This is a convenient mechanism of returning values of parameters from a procedure. In order to reach the parameters declared in a procedure with "Var", the parameter address must first be read. The address is then used to access the parameter. The example below demonstrates accessing a parameter "C" passed by name to a procedure "XYZ".

```
Procedure XYZ(VarA,B,C:integer);
begin    .
#A
 LDY #3
 LDA (SP),Y; GET LSB OF ADDRESS OF C
 STA T; SAVE IN PAGE ZERO TEMP
 INY
 LDA (SP),Y; GET MSB OF ADDRESS OF C
 STA T+1
 LDY #0
 LDA (T),Y; GET LSB OF C
 STA ..
 .
 .
 .
#
end;
```

In the above example the offset from the stack pointer to the address of C is:

```
     3 bytes offset from (SP) to first local variable
     0 local variables
     3 bytes total to the LSB and 4 bytes to the MSB of the address
```

The address is then put in the zero page temporary T and indirect references are then made to C with T.

Predefined Labels

The following table gives the absolute locations of the predefined labels SP, LOCAL, and T. SP and LOCAL contain the addresses of the bottom and top of the Pascal variables stack, respectively. T is the start of the temporary registers. There can be up to 16 temporary labels going from T to T+15.

```
SP          EQU  4
LOCAL       EQU  2
T           EQU 16
```

Manual Errata

The table in the manual showing the number of bytes of memory provided on the stack for each type of variable or constant contains an error. Char and Boolean types are provided 1 byte of memory rather than the 2 bytes shown in the table. Similarly, the Value Parameter (Char,Boolean) should be 1 byte.

New Assembly Language Routines

You can add the following routines to your library of Pascal extensions. To use them: 1) type the program using the Kyan Pascal editor and save the file; 2) include the routine ("# filename") in the declarations section of your Pascal program; and 3) call the routine as needed in your Pascal program.

Random Number Generator

Filename: RANDOM.I
Function: returns a real number between 0 and 1.
Assembly Language Listing:

```
FUNCTION RANDOM:REAL;
BEGIN
#A
 TXA
 PHA
 LDA #0
 STA T
RAN1 INC T
 JSR POLY
 CMP #0
 BEQ RAN1
 ORA #$10
 LDY #3
 STA (SP),Y
;
RAN2 INY
 JSR POLY
 ROL A
 ROL A
 ROL A
 ROL A
 AND #$F0
 STA T+1
 JSR POLY
 ORA T+1
 STA (SP),Y
 CPY #9
 BCC RAN2
 LDA T
 INY
 STA (SP),Y
 PLA
 TAX
#
END;
#A
POLY TYA
 PHA
 LDY #0
POLY1 INY
 CLC
 ROL POLYN
 ROL POLYN+1
 ROL POLYN+2
 ROL POLYN+3
 ROL POLYN+4
 ROL POLYN+5
 ROL POLYN+6
 ROL POLYN+7
 BCC POLY3
;
 LDX #0
POLY2 LDA POLYN,X
 EOR GEN,X
 STA POLYN,X
 INX
 CPX #8
 BCC POLY2
 SEC
;
POLY3 ROL T+2
 CPY #4
 BCC POLY1
;
 PLA
 TAY
 LDA T+2
 AND #$0F
 CMP #$0A
 BCS POLY
 RTS
;
;
GEN DW $A1
 DB $A2
 DB $1A
 DB $A2
 DB $91
 DB $C3
 DB $93
 DB $C0
;
POLYN DW $63
 DB $42
 DB $A1
 DB $23
 DB $55
 DB $09
 DB $03
 DB $87
#
```

If you want to test the "randomness" of RANDOM.I, try the following Pascal test program. It computes the mean and standard deviation of random numbers generated by the RANDOM.I routine.

```
PROGRAM TESTR;
VAR N,I:INTEGER;
 X,Y,Z:REAL;
#I RANDOM.I
BEGIN
 REPEAT BEGIN
 WRITE('NUMBER OF RANDOM·NUMBERS = ');
 READLN(N);
 Y:=0;
 Z:=0;
 FOR I:= 1 TO N DO BEGIN
    X:=RANDOM;
    Z:=Z+X;
    Y:=Y+ (0.5-X)*(0.5-X);
 END;
WRITELN('MEAN (1/2)= ',Z/N:15,'   VARIANCE (8.33)= ',Y/(N-1):15);
END;
UNTIL N=2;
END.
```

Go To X,Y Routine

Filename: GOTOXY.I
Procedure: positions the cursor on the screen at location X,Y.
Assembly Language Listing:

```
PROCEDURE GOTOXY (X,Y:INTEGER);
BEGIN
#A
 LDY #3
 LDA (SP),Y
 TAY
 DEY
 STY $25
 LDY #5
 LDA (SP),Y
 TAY
 DEY
 STY $24
 JSR $FC22
#
END;
```

If you want to test GOTOXY.I, try the following Pascal test program. It will ask a question at location (X,Y) and then print the answer at location (X+6,Y+3).

```
PROGRAM TESTGO;
VAR X,Y,Z:INTEGER;
#I GOTOXY.I
BEGIN
 X:=15;Y:=10;
 GOTOXY (X,Y); WRITELN('ENTER VALUE ');GOTOXY (X+12,Y);
 READLN (Z); GOTOXY(X+6,Y+3); WRITELN ('ANSWER ',Z);
END.
```

Letters to the Editor

".... can you recommend a good Pascal programming book which can be used to supplement the Kyan Pascal tutorial manual?"

Programming in Pascal, P. Grogono, Addison-Wesley Publishing, 1978.

Pascal, A Problem Solving Approach, E.B. Kaufman, Addison-Wesley Publishing, 1982.

Introduction to Pascal, R. Zaks, Sybex, Inc. 1981.

Pascal User Manual and Report, K. Jensen and N. Wirth, Springer-Verlag, 1974.

"... I want to begin using assembly language routines in my Pascal programs. Are there any books you would recommend?"

Programming a Micro-computer: 6502, C.C. Foster, Addison-Wesley Publishing, 1978.

6502 Assembly Language Programming, L.A. Leventhal, Osborne/McGraw-Hill, Inc. 1979.

"... I want to learn more about ProDOS, but none of the stores in my area carry the Apple manuals. Can you help?"

Until quite recently, Apple maintained a dealer policy which made it very difficult for bookstores to carry Apple manuals. However, they have now given book distribution to McGraw-Hill, and you will begin to see more technical reference manuals for Apple products in your local stores. In the meantime, you can order a copy of the ProDOS User's Manual and the ProDOS Technical Reference Manual directly from the McGraw-Hill Bookstore, 1221 Avenue of the Americas, New York, NY 10020. You can also order the books by phone (212-512-6230) using a major credit card.

"... after running a graphics program, I have a lot of garbage on the screen and the software won't work properly. What is the problem?"

After running a graphics program, your Apple is still in the hi-res graphics mode. To get out of hi-res, simply press <control><reset>. The ProDOS prompt will then appear and you can enter any system filename (e.g., ED, E80, or your program name if you want to run it again).

We welcome letters from Kyan Pascal programmers. You can send us questions, programming hints, gripes, new product ideas, or anything else which might be of interest to other Kyan Pascal owners. We will print as many as possible in this space.

Send your letters to:

Update .... Kyan
Kyan Software
1850 Union Street, #183
San Francisco, CA 94123

December 31, 1985

Dear Friend:

Since the introduction of Kyan Pascal in Spring 1985, we have received many comments, suggestions, and not a few complaints from our users. Well, we want you to know that we listen. And, as a result of your input, we are introducing a totally new version of Kyan Pascal, three programming toolkits, and a new 6502 Macro Assembler/Linker.

Version 2.0 of Kyan Pascal is described in the Kyan Pascal section of this newsletter. The toolkits and Assembler/Linker are described in the "What's New" section. We are quite proud of these new additions to the Kyan Software product family and believe you will find them to be extremely useful.

We are especially proud of the new UNIX-like operating environment in Kyan Pascal which we call KIX™. With KIX™ we have turned ProDos into a true operating system. Now you can forget the Filer. Simple command line arguments entered at the system prompt allow you to alter files, work with directories, dump screens to the printer, and more. If you have worked with UNIX before, you know about the speed and convenience of this powerful operating system. If you haven't get ready for an exciting new experience.

In keeping up with our police of low cost upgrades for existing owners, we are offering all registered owners of Kyan Pascal Version 1.- the right to upgrade to Version 2.0 of Kyan Pascal (including a completely new manual) for only $20.00 ($10.00 if you have purchased Kyan Pascal after 12/1/85, and no charge if you purchased it after 1/1/86). We are also offering a 15 percent discount on the Toolkits and Macro Assembler/Linker.

We hope to continue bringing you an expanding line of new programming software. We truly appreciate your support and welcome your comments and suggestions. Everyone at Kyan Software extends to you our Best Wishes for a happy and prosperous New Year.

Sincerely,

Thomas E. Eckmann
President
Kyan Software, Inc.

kyan software 1850 union street #183 san francisco, california 94123 (415) 775-2923

UPDATE ..... KYAN!

Kyan Software, Inc.
1850 Union Street, #183
San Francisco, CA. 94123

Volume 1 Number 2

January/February Issue

APPLE EDITION

WHAT'S NEW

Kyan Software, Inc. has three great new toolkits for you: a **Programming Utility Toolkit**, an **Advanced Graphics Toolkit**, and a **MouseText Toolkit**. In addition we are introducing a new **Macro Assembler/Linker**. Each of these products is described in the following section.

Programming Utility ToolKit

The **Programming Utility Toolkit** gives you a large library of file management and programming utility routines. The Toolkit includes utilities which allow you to access the disk from within programs and routines which make it easier to convert UCSD and Turbo-Pascal programs into ProDOS-based Kyan Pascal programs. In addition, the Toolkit features an advanced random number generator, Turtle graphics, sound routines, screen and cursor control routines, quick sort routines, ASCII character to numeric conversion routines, and more. The **Programming Utility Toolkit** routines are easy to use. Simply "include" them in your Pascal source code. The Kyan Pascal compiler does the rest.

The Programming Utility Toolkit will run on any Apple II with 64K of memory and any version of Kyan Pascal for the Apple II.

Advanced Graphics ToolKit

The **Advanced Graphics Toolkit** lets you add stunning hi-res and double hi-res graphics to your Kyan Pascal programs. It contains a complete set of graphics primitives which allow the user to develop custom graphics. It also contains an extensive library of standard graphics routines and procedures. With the **Advanced Graphics Toolkit**, even novice Pascal programmers can create programs with a sophisticated user interface and exotic graphics displays.

The graphics primitives are a set of fundamental procedures which are used to create graphic displays. The primitives include line and text drawing, device information, and initialization commands necessary to create custom graphics. Graphics are created by calling the primitive commands and specifying desired parameters (e.g., line length, start/finish point, etc.). By developing and saving custom graphics, programmers can build personal libraries of graphics routines.

The **Advanced Graphics Toolkit** also features a library of Pascal and assembly language source code routines which can be called to perform "standard" graphics functions. These "standard" routines, however, are far from commonplace. For example, with the **Advanced Graphics Toolkit**, you can create a two or three dimensional object; draw a parallel or perspective projection of this object; and then, scale, rotate, or translate the object into different forms. The Toolkit library also contains windowing and clipping routines which allow you to select and enlarge portions of a drawing (windows) and "clip" away the rest. Finally, the library includes routines for hidden surface and line removal, shading, and the genertion of curves.

The **Advanced Graphics Toolkit** requires an Apple IIc or IIe (with extended 80 column card) and Version 2.0 of Kyan Pascal.

MouseText Toolkit

The **MouseText Toolkit** lets you add a Macintosh-like user interface to your Kyan Pascal programs. Pull-down menus, windows, and mouse-controlled cursor movements can be integrated in your Apple II programs with ease. Now, you can take full advantage of the capabilities of the Apple IIc and Apple IIe (with updated character ROM).

The **MouseText Toolkit** consists of software routines for mouse-operated menus and text windows. The routines utilize the MouseText icons stored in ROM on the Apple IIc and updated IIe and support such functions as: cursor selection and display, menu bar displays, and menu item selection; window selection and display, window dragging and size changing, and writing text in windows; and scrolling through documents.

With Kyan's **MouseText Toolkit**, you can write Apple II programs which have a desktop environment. Windows are used to represent documents on the desktop. With Toolkit routines, you can open and close windows, move windows around, and reduce or enlarge windows. You can also write programs which are event-driven (i.e., they respond to mouse, keyboard, update, or application events).

The **MouseText Toolkit** makes it easy for even novice Pascal programmers to create software with state-of-the-art features. It requires an Apple IIc or Apple IIe (with updated character ROM and extended 80 column card) and Version 2.0 of Kyan Pascal.

Kyan Macro Assembler/Linker

The **Kyan Macro Assembler/Linker** is a powerful ProDOS-based programming tool which allows you to create assembly language programs that can run on any Apple II. It features a full-screen text editor, 6502 macro assembler, object module linker, and Kyan's unique KIX[tm] operating environment. With the **Kyan Macro Assembler/Linker**, assembly language programming has never been easier!

The **Kyan Macro Assembler/Linker** includes:

- o An extremely fast object module linker;
- o Kyan's full-screen text editor with support for 40 and 80 column screens, upper and lower case letters, and a full range of text editing functions;
- o A 6502 macro assembler which produces relocatable binary files optimized for speed. The assembler supports 65C02 opcodes, local variables, identifiers up to 255 characters in length, calls from BASIC, and a full library of error messages. It also supports cross-referencing of variable names;
- o The KIX[tm] operating environment which features a UNIX-like command structure for ease of use. With KIX[tm], programmers can call directories, alter files, and execute a wide range of other file managment commands directly from the system prompt. Forget the ProDOS Filer; now the Apple has a real operating system!

The **Kyan Macro Assembler/Linker** is not copy-protected and includes a complete reference manual. It will run on any Apple II with 64K of memory and a single disk drive. It is fully compatible with all of Kyan's Pascal toolkits.

Price and Availability

These new products are scheduled to be available on February 1, 1986 (Please forgive us if we are a week or two late).

Each package will be offered to registered owners of Kyan Pascal at a reduced price.

	Suggested Retail Price	Special Offer Price*
Programming Utility Toolkit	49.95	42.95
Advanced Graphics Toolkit	49.95	42.95
MouseText Toolkit	49.95	42.95
All 3 Toolkits	149.85	128.85
Kyan Macro Assembler/Linker	69.95	59.95

Plus shipping .

* Special Offer Price expires April 1, 1985.

KYAN PASCAL UPDATE

KYAN PASCAL -- VERSION 2.0

Version 2.0 of Kyan Pascal is a complete rewrite of the Pascal compiler and assembler. Many new features and capabilities have been added to make Kyan Pascal a full-featured software development system. In Version 2.0 we have incorporated many of the suggestions received from current users and have added a range of enhancements developed by the Kyan staff.

One of the most significant changes in Version 2.0 is a completely new users manual. This manual has an expanded tutorial section, comprehensive Pascal reference section, and several new sections on Pascal and assembly language programming. The manual is bound in a 3 ring binder for ease of use and convenient updates. Changes in the software include:

o Full compatibility with the ISO standard for Pascal. Version 2.0 has successfully compiled and assembled the Pascal test suite used by the Federal Software Testing Center to validate and certify Pascal compilers.

o Improved error handling. Version 2.0 includes an expanded list of compiler, assembler and runtime error messages.

o Intermediate assembly language files. Version 2.0 lets you capture the assembly language output of the Pascal compiler as a text file. You can then edit or modify this file prior to assembling.

o New procedures: Page, Close, Pack and Unpack.

o New data type: String

o New Library Functions: KEYPRESS, GETCHR, GOTOXY,RANDOM

Last, but far from least, is Kyan Software's new operating environment called KIX[tm]. KIX[tm] features a UNIX-like command structure which allows you to: execute file management commands; list and edit directories; dump screens to a printer; and, perform dozens of other functions; all from a command line prompt. The limitations and inconvenience of the ProDOS Filer are now history. With KIX[tm] and Version 2.0, you'll think you are programming on a million dollar mainframe!

Price and Availability

In keeping with Kyan's policy of low cost upgrades, Version 2.0 is available to all registered owners of Kyan Pascal (Version 1.-) for only $20.00 ($10.00 if you purchased after December 1, 1985, and no charge if you purchased after January 1, 1986) plus freight. This price includes both the new software and manual. To be eligible for the upgrade, you must complete the enclosed form and mail it to Kyan Software with your payment and Kyan Pascal source disk. We will begin shipments of upgrades in the middle of February.

Advanced Pascal programming: The Chain feature

By using the "Chain" feature in Pascal programs, the programmer can call and run compiled programs from within an original program. Variables can also be passed from the original to the chained program.

To use the chain feature, first compile both the original and the chained program. Then, when the computer reaches the statement **Chain ('FILENAME.0')** in the original program, the computer will pass control to the first statement in the chained program. Essentially, chaining is telling the computer to RUN the specified object program.

Variables used in the first program may be lost when the second program is called. To saved the variables, they must be passed from your original program to the chained program. To do this, the variables must be: declared in the second program inthe same order as they were in the original program; and, they must be of the same type. The example programs below demonstrate the method of passing parameters between programs.

First (original) program:

```
PROGRAM Retail (Input, Output);
  TYPE
     String=ARRAY[1..64] OF Char;
  VAR
     ProductName: String;
     Price: Real;
  BEGIN
   WRITELN(CHR(12));
   WRITELN('Retail salesperson, what is the name');
   WRITE ('of the product you sold?');
    READLN(ProductName);
   WRITELN('And what is the price, in dollars');
   WRITELN('and cents, that you sold the');
   WRITE (ProductName,'for?-->$');
    READLN(Price);
  CHAIN('/DIR/PROFIT.0')
 END.
```

Second (chained) program:

```
PROGRAM FindProfit(Input,Output);
 TYPE
    String: ARRAY[1..64] OF Char;
 VAR
    ProductName: String;
    Price, Cost, Profit: Real;

BEGIN
  WRITELN(CHR(12));
  WRITELN('>You have chained to the second program<');
  WRITELN('What was our cost of the',ProductName');
  WRITELN('that you sold?-->');
    READLN(Cost);
  WRITELN('Okay, salesperson, you sold a');
  WRITELN(ProductName,' for $',Price:4:2);
  WRITELN('It cost us $',Cost:4:2,' so we');
Profit:=Price-Cost;
  WRITELN('Made a profit of $',Profit:4:2)
END.
```

How the two programs work:

The first program, "Retail," assumes that the user is a retail salesperson who has just sold a product to a customer at a specific price. The computer asks the salesperson for the product's name and the price at which it was sold. When he responds, the computer runs the second program (via **Chain**), called **FindProfit**. The product name and price are passed to **FindProfit**. The program then asks for the cost of the product (which should be less than the price it was sold for). The program then gives a rundown of past events and computes the profit (with the formula **Profit:=Price-Cost**). Finally, it shows the profit made.

Points to remember about chaining:

When chaining:

- o The filename must specify the object code file (must have the .O extension unless it was renamed.
- o No statements after the Chain statement will be executed (unless the second program chains back to the first under special conditions).

When passing parameters:

- o The variables must be declared in the same order in both programs.
- o The data type of the variables must remain the same.

How Kyan Pascal uses these passed parameters:

- o Variables are stored starting at high memory and grow towards low memory.
- o Variables are stored on the stack.

The Kyan Text Editor

Many people have written to us regarding the Kyan text editor. Perhaps a little background would help you understand the philosophy behind our editor and why you have a choice in the type of editor you use with Kyan Pascal.

First, let's define what we mean by the term text editor. Text editors are simply word processing programs. In Kyan Pascal an editor is used to enter Pascal source code into the computer and to create ASCII text files which can be read by the Pascal compiler.

Text editors are undoubtedly the most popular and widely-used class of software for the personal computer. There are many different editors available, and you have a wide selection to choose from. Everyone has a favorite editor which they believe to be the "best" one available. If I had to guess which is your favorite, I would say it's the one you first used when you were learning to use your computer. It's human nature -- people generally like the things they know best.

In the early stages of development of Kyan Pascal, there was a great deal of debate over the design of the text editor. One school of thought was that we should not have any text editor at all. Since our compiler can read any sequential ASCII file, it was argued that we should let users supply their own text editor for writing Pascal source code. The other school of thought was to supply a text editor which was adequate for writing and editing Pascal source code, but which did not include the bells and whistles found in stand-alone editors.

This second school of thought prevailed. We decided to include an editor which was modeled after those found on mainframe computers. This would allow students just learning the Pascal language to more easily make the transition from their micros to mainframe computers found at most schools.

The Kyan editor is modeled after Wordstar. The control key command structure of the editor is widely used on larger computers and with the professional level language compilers. If you know and like Wordstar, you probably like the Kyan editor. If you don't, then you may object to it. Regardless of which camp you are in, just remember one thing ..... **the Kyan Pascal compiler is compatible with any text editor which generates sequential ASCII files.**

One way to work with Kyan Pascal is to write your source code program using your favorite word processing software. Then, reset the computer, load the Kyan Pascal compiler, and compile your program. The Kyan editor can then be used for simple edits of your program.

My favorite way to write Kyan Pascal programs is to use the word processor in Appleworks to write my programs. I then compile the program with Kyan Pascal. When the compiler lists my errors, I use the Kyan editor to do the quick patch jobs. It works wonderfully!

PASCAL PROGRAMMING

Several users have submitted utility programs to **Update....Kyan**, which we would like to pass on to you. We can't guarantee that these programs are useful in your application; but you may want to examine and experiment with them.

Useful Routines

The Jensen/Wirth Standard for Pascal does not include cursor controls and other procedures many users may want. Calvin Glomb sent us a whole set of procedures for use in the 80 column mode, which he says are derived mostly from Apple's 80-Column Text Card Manual. These routines should help a lot of people. (Does anyone out there have similar routines for the 40 column mode? They, too, would be greatly appreciated).

BEEP — produces a 1000 Hz tone for 0.1 seconds.

```
procedure beep;
begin
   write(chr(7));
end;
```

LEFT — moves the cursor left one character.

```
procedure left;
begin
  write(chr(8));
end;
```

DOWN moves the cursor down one character.

```
procedure down;
begin
  write(chr(10));
end;
```

CLREOS clears the screen from the cursor to the end of the screen.

```
procedure clreos;
begin
  write(chr(11));
end;
```

CLEAR clears the entire screen.

```
procedure clear;
begin
  write(chr(12));
end;
```

NORMAL sets the cursor format to normal.

```
procedure normal;
begin
  write(chr(14));
end;
```

INVERSE sets the cursor format to inverse.

```
procedure nverse;
begin
  write(chr15));
end;
```

SCROLLDOWN scrolls the display down one line and leaves the cursor in the current position.

```
procedure scrolldown;
begin
  write(chr(22));
end;
```

HOME — returns the cursor to the upper left corner of the screen.

```
procedure home;
begin
   write(chr(25));
end;
```

CLRLIN — clears the line the cursor is on.

```
procedure clrlin;
begin
   write(chr(26));
end;
```

RIGHT — moves the cursor to the right one character.

```
procedure right;
begin
   write(chr(28));
end;
```

CLREOL — clears the line from the current cursor position.

```
procedure clreol;
begin
   write(chr(29));
end;
```

UP — moves the cursor up one character.

```
procedure up;
begin
   write(chr(31));
end;
```

GOTOXY — moves the cursor to the position on the screen specified by COL and ROW, with COL having a range 0 - 70 and ROW having a range 0 - 23

```
procedure gotoxy(col,row: integer);
var
   go: integer;
begin
   home;
   go:=0;
   while col > go do begin
        right;
```

```
        go:= go + 1;
        end;
    go:= 0;
    while row > go do begin
        down;
        go:= go + 1;
        end
end.
```

PAUSE will cause a delay of a time period equal to the variable time. This period is not specified but one second is about equal to 1000.

```
procedure pause(time:  integer);
var
   loop: integer;
begin
   for loop:= 1 to time do
end;
```

W. P. Hudson, of Corpus Christi, has also sent us another version of these functions, as well as some extra goodies like FLASH.

```
Procedure CLRSCRN;   (* Clears screen like BASIC 'HOME'  *)
Begin
*a
        JSR $FC58
*
end;

Procedure INVERSE;    (* Sets cursor format to inverse mode *)
var
   locateI: ^integer;
Begin
   assign(locateI,50);
   locateI^:= 63
end;

Procedure NORMAL;     (* Sets cursor format to normal mode *)
var
   locateN: ^integer;
Begin
   assign(locateN,50);
   locateN^:= 255
end;

Procedure FLASH;      (* Flashes cursor between inverse and
var                                normal modes *)
   locateF: ^integer;
Begin
   assign(locateF, 50);
```

```
   locateF^:= 127
end;

Procedure VTAB(locv:integer);    (* Moves cursor to position X *)
var
   locatev: ^integer;
Begin
   assign(locatev,37);
   locatev^:= locv;
Begin
*a
        JSR $FC22
   end
end;

Procedure HTAB(loch: integer);    (* Moves cursor down to line Y *)
var
   locateh: ^integer;
Begin
   assign(locateh,36);
   locateh^:= loch
end;

Procedure LOCATE(locv,loch: integer);   (* Moves cursor to
var                                                position XY *)
   locatev, locateh: ^integer;
Begin
   assign(locatev,37);
   locatev^:= locv;
begin
*a
        JSR $FC22
*
    end;
   assign(locateh,36);
   locateh^:= loch;
end;
```

As Mr. Hudson observes, Procedure LOCATE is similar to the assembly language "GOTOXY" routine version in the last issue of **Update** ..... **Kyan**.

Print Program with Line Numbers

Erik Warren has developed a program that will permit you to add line numbers to a printout of your source code listing. Line numbers can be useful when debugging a program.

```
Program Print(Input,Output)

Var
   f: text;
   s: array [1..64] of char;
   n, x: integer;
c: char;

*i pr.i

Begin
x:=0;
n:=1;
write('Pathname?  ');
   readln(s);
write('Line numbers (y/n)?  ');
   readln(c);
reset(f,s);
pr(1);  (* printer slot  *)

while not eof(f) do
   begin
   if (c='y') and (x=0) then
        begin
        x:=1;
        if n<10 then write ('000',n,'  ');
        if (n>9) and (n<100) then write ('00',n,'  ');
        if (n>99) and (n<1000) then write ('0',n,'  ');
        if n>999 then write (n,'  ');
        end;  (* if c and x  *)
   if eoln(f) then
   begin
   x:=0;
   n:=n+1;
   writeln;
   end  (*  if eoln  *)
  else
   write(f^);
  get (f)
end;  (*  while  *)
writeln;
pr(0)  (*  screen  *)
end.
```

ASSEMBLY LANGUAGE PROGRAMMING

Some Useful Utilities

Keith Symon of Madison, Wisconsin sent us some useful utilities. He notes that some of these procedures use peek(loc) and poke (loc,val) from the first issue of **Update.....Kyan**. Some of these procedures perform the same functions as procedures in the Pascal Programming section.

Note: All assembly language source codes must be in upper-case letters.

```
Procedure CALL(loc: integer);     (*Jump to specified memory location*)
Begin
*a
        LDY #3                    ;loc
        LDA (SP),Y
        STA ADD+1
        INY
        LDA (SP),Y
        STA ADD+2
ADD     JSR $BE9E                 ;called location
#
End;

Procedure HOME;                   (*Moves cursor to upper left-hand corner*)
Begin
        CALL(-936)
End;

Procedure HTAB(I: integer);       (*Moves cursor to specified horizontal
Begin                               position*)
  POKE(36,I)
End;

Procedure VTAB(I: integer);       (*Moves cursor to specified vertical
Begin                               position*)
  POKE(37,I);
*a
  JSR $FC22;     VTAB set
#
End;

Procedure CLEARLINE;              (*Clears line*)
Begin
  CALL(-868)
End;
```

```
Procedure GETCHAR(Var ch:char); (* like GET in BASIC *)
Begin
*a
        LDY #3              ;get ch character
        LDA (SP),Y
        STA T
        INY
        LDA (SP),Y
        STA T+1
        JSR $FD0C           ;rdkey
        AND #$7F            ;remove hi bit
        LDY #0
        STA (T),Y
        JSR $FDED           ;cout
*
End;

Procedure SOFTGHT; (* switch to hires, do not clear screen *)
Begin
*a
        LDA $C053
        LDA $C054
        LDA $C057
        LDA $C050
*
End;
```

Programming Note.....The X Register

We frequently get calls from programmers whose latest creation keeps crashing. The culprit is frequently found to be the X register. The 6502 X register is used by the compiler as a stack pointer. *If you use the X register in your own assembly language routine, you must save and restore it.*

UPDATE ... KYAN

Kyan Software Inc.
1850 Union Street #183
San Francisco, CA 94123

March/April Issue
Volume 1, Number 3

Apple Edition

(Editor: Sonja Newell)

WHAT'S NEW?

INTRODUCING ... KIX™

KIX is a powerful UNIX-like operating shell which works with ProDOS to give the programmer direct access to and full control over all files, volumes and directories in the system. **KIX** eliminates the need for the Filer. It provides you with a broader range of disk management capabilities and is much easier to use because you don't need to wade through a long series of menus. Anytime the system prompt is present, you can call and execute a KIX command. **KIX** consists of more than 25 commands that let you find, move, copy, compare, and manipulate files and directories on your disks.

To give you a better idea of why we are so excited about this new development, we'd like to briefly describe the six groups of **KIX** commands.

Directory Control

ProDOS stores files in volume directories and subdirectories. With **KIX** you have four simple commands which allow you to print, create, change or delete these directories.

Listing Directory and File Contents

With three simple commands, KIX allows you to list, print and/or concatenate files and directories. Each command has a series of options which let you specify exactly what information you want listed. For example, you can print a simple listing of disk directories which tells you only the name for each file or directory and the number of blocks free, or, you can print a complete listing which displays the length, protection status, file type, starting address and other important file information.

Manipulating Files, Directories, and Volumes

The six commands in this category allow you to copy, move, delete, or change the protection status of files and directories. They also allow you to format and copy new volumes (including the new 3.5" Unidisk).

Comparing Files and Volumes

The two commands in this category allow you to compare files and volumes to determine if they match. If two volumes do not match, the block and byte location of the first difference will be reported. If two text files do not match, the lines containing the differences will be printed.

Searching Files and Directories

One command searches all volumes and directories for a specified file (i.e., "I put that file in a subdirectory which made sense at the time, but now, ..."). Another command allows you search text files for a string of characters (e.g., find all the text files which contain a reference to Kyan Software).

Date and Display Attributes

This category contains nine utility commands which allow you to perform such activities as setting the system time and date, switching to/from 40 or 80 column mode, dumping the contents of the screen to the printer, and more.

KIX also supports the use of wildcards. They can be used with various KIX commands to further expand the power and flexibility of the KIX system.

KIX is included in Version 2.0 of **Kyan Pascal** and the **Kyan Macro Assembler.** It will also be available soon as a stand-alone product which will work within the framework of AppleWorks and other popular Apple II software. Many of the KIX commands are available to Kyan programmers who use the Programming Utility Toolkit. The Toolkit procedures allow you to write disk management routines into your own programs and applications software.

NEW DEVELOPMENTS IN APPLE II HARDWARE

The 65C816 is a powerful new microprocessor developed by the Western Design Center. The engineers involved in the design of the new processor were also involved in the original development of the 6502 processor.

The 65C816 has been in development for several years and was rumored to be the processor around which Steve Wosniak was designing the mysterious and ill-fated Apple IIX. The chip was delayed in development and apparently contributed to the demise of this product at Apple. The chip has now been debugged and manufacturing rights have been licensed to several companies.

The 65C816 microprocessor has the advantage of operating in three different modes: (1) 6502 emulation mode which allows you to run all existing Apple II software; (2) an accelerated 6502 mode which allows you to run some Apple II software at a faster clock rate; and, (3) a true 16-bit mode which can run a new generation of software at speeds of 3 mHz or more. This processor can also directly address more than one megabyte of memory (versus 64K for the 6502). The capabilities offered by this new processor hold out the promise of a whole new generation of high performance Apple II hardware and software.

The first commercial application of this processor is an Apple II add-on board manufactured by Checkmate Technologies of Tempe Arizona. Checkmate is marketing the board as a plug-in adjunct to its RAM expansion boards. Since the Checkmate board is a plug-in device which uses the standard Apple I/O, it runs at a clockrate of only 1 mHz (in all modes). However, the Checkmate board offers hobbyists and professional developers a first opportunity to play around with this exciting new processor and explore the new opportunites which it affords for a new generation of Apple II software. (Note: As of this date, a disk operating system is still not available to support true 16 bit operation of the processor. Checkmate expects to have one available very soon, but for now, users are limited to running the chip in 6502 emulation mode using ProDOS).

(Next Issue: RAM cards and Static RAM with 10 year battery backup!)

INTRODUCTION TO HIGH LEVEL LANGUAGES

by the Kyan Staff

With this issue of **Update...Kyan**, we are beginning a series of articles which describe the origins and evolution of the Pascal programming language. This series starts with an overview of programming languages in general and where Pascal fits in. In the next issue, we will discuss computer hardware and how it works with the software you write. Subsequent issues will talk about software engineering, debugging and other topics of general interest.

If you have a particular topic you would like discussed in this column, please let us know. We are writing this column for you and we would like to hear from you.

Rather than re-invent the wheel, we will use, when appropriate, excerpts from popular books or periodicals. This month, we have an excerpt from the book **"Oh! Pascal!"**, by Doug Cooper and Michael Clancy (second edition; W. W. Morton, New York, NY 1985) This is one of the best textbooks available for Pascal programming. The authors have a wonderful writing style, and we highly recommend it.

****

"There's and old story about an untutored bumpkin who listened to some students as they talked about the stars. Although the concepts they discussed were strange and new, he felt he could understand how astronomers used telescopes to measure the distance from the earth to the celestial bodies. It even seemed reasonable that they could predict the stars' relative positions and motions. What totally puzzled him, though, was how the devil they were able to find the stars' names.

"People sometimes approach programming languages in the same way,-- as though they're complicated mathematical codes that the first computer scientists were lucky enough to break. Well, they are ciphers of a sort, but they're not so hard to crack. Let's look at the three basic kinds of programming languages -- low-level machine languages, intermediate assembly languages, and finally, high-level languages like Pascal.

"The most basic programming codes belong to an instruction set. These are the computer's built-in commands, and they aren't much more sophisticated than the operations we can punch into a programmable hand calculator. There are instructions for doing simple arithmetic, of course, and for saving answers and values as we go along. There are usually a variety of instructions available for comparing values, and for deciding what to do next. A special set of instructions store and retrieve things from the computer's memory. The fanciest instructions usually deal with getting more instructions.

"A machine language -- the simplest programming code -- is defined by numbering the basic instructions. When we do this, each instruction's number becomes its code name. If we use eight-digit binary values for numbering (as computers often do), we can name 256 different instructions, starting with 00000000 and ending with 11111111. A machine language program is nothing but a long series of eight digit numbers.

"Machine language programming is easy, but it's incredibly tedious. Fortunately, one of the earliest programmers had a bright idea. Why not write a machine language program that could recognize short sequences of English letters, and would automatically translate the English into the proper machine language instructions? Why not, indeed! Such programs were called assemblers; they understood assembly language, and were soon found on every computer.

"Assembly language programming is a bit more palatable. An assembly language program is a sequence of two-to-four-letter assembly language commands, often accompanied by an additional shorthand that identifies locations in the computer's memory. For instance, the assembly command

ADD R2,R4 means 'add the contents of memory location R2 to memory location R4, and save the sum in R4.' It isn't hard to imagine carrying out the same kind of command on any hand calculator that has a built-in memory.

"Although assembly language was a convenience, it hardly exhausted the limits of human ingenuity. Why be limited to three-letter words? Programmers wanted to express themselves in relatively English sentences, rather than in the computer-oriented terms of machine and assembly languages. In response, research teams did the obvious. They repeated the same step that had led to assemblers, and wrote more complicated programs, called interpreters and compilers. The new programs translated increasingly sophisticated sequences of letters into a form the computer could understand. The letters, and the words they formed, were called high-level languages.

"High-level languages, like FORTRAN, BASIC, and Pascal, are consciously designed to help solve problems. In contrast, low-level machine and assembly languages, were expressly intended to operate computers. Programers who use high-level languages don't have to worry about getting instructions, or keeping track of numbered memory locations. Instead, they give commands in a language that usually resembles a terse English. A typical high-level language's program consists of phrases that are liable to be found in the statement of a solution – if a condition is met then we take an action or else do something different.

"Many programming languages have been designed in response to different problem-solving requirements. Just as there are several types of hand calculators (statistical, business, scientific), there are also job-specific computer languages. You can write most programs using any language, in the same way that you could use a financial calculator to solve a statistics problem. However, it makes sense to use the most appropriate tool. We can discover the original purpose of some languages from their names:

> **FORTRAN: FOR**mula **TRAN**slator is one of the earliest and most widespread languages. It's intended mainly for scientific applications.
>
> **COBOL: CO**mmon **B**usiness **O**riented **L**anguage was developed as a standard for business computing. Many COBOL instructions are designed specifically for payroll or accounting applications.
>
> **BASIC: B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode is a simple language that's used to teach basic computer applications. Although it's easy to learn, BASIC doesn't go very far. It's a poor basis for understanding programming, and is no longer widely taught in college level programming courses.
>
> **LISP: LIS**t **P**rocessing language is widely used in programs that involve processing of symbols, from mathematical symbols to the symbols that form natural (spoken) languages. It's one of the main languages used in artificial intelligence research.

"Pascal was named after the 17th century mathematician and religious zealot Blaise Pascal. Since it's not an acronym (it doesn't stand for anything) only the first letter is capitalized. Pascal was created with two main goals:

1. To provide a teaching language that would bring out concepts common to all languages, while avoiding inconsistencies and unnecessary detail.

2. To define a truly standard language that would be cheap and easy to implement on any computer.

"In a sense, Pascal is a *lingua franca*, or common tongue, of programming. It's easy to learn, and provides an excellent foundation for learning other languages. We've found that people who know Pascal can master BASIC in an afternoon, and pick up FORTRAN in a week or two.

"But what does Pascal look like? Niklaus Wirth, the Swiss professor who designed the language, intended that Pascal be clear, readable, and unambiguous as possible. Reserved words are the bare bones Pascal language. Clearly, Pascal isn't written in binary code. In fact, we're reminded of the famous ad:

If u cn rd ths ad, u cn gt a gd jb--Learn Speedwriting!

"One of Pascal's big advantages over some earlier languages is that it lets us write a program using almost exactly the same terms we used to state the original problem's solution. Some day soon we expect to be seeing this sign in the subways:

```
if YouCanReadThis then
begin
  StartWork(Soon);
   Earn(BigBucks)
end;
```

"Is Pascal the ultimate programming language? No. An eventual successor to Pascal may be Modula-2, which was also developed by Wirth. Don't worry that you're studying the wrong language, though, because the first term's worth of Modula-2 is almost exactly like Pascal. Modula-2 contains some additional features that make it attractive for later programming courses, particularly those that involve writing large programs that translate languages (the compiler course) or control computers (the operating systems course). Within a few years, Modula-2 may become a language widely used for undergraduate coursework.

"Another potential Pascal successor, the Ada programming language, was commissioned by the U.S. Department of Defense. The DoD hoped to create a language that would be more reliable for the control of weapons systems, as well as less expensive to program in, than FORTRAN and the others. At the time of this writing it is not clear that either goal has been met, nor that the language will be widely used outside of defense contractors' programming shops."

****

Thank you, Professors Cooper and Clancy! The preceding text is an excerpt from their textbook ***"Oh! Pascal!"*** *. In the next issue of* ***Update ... Kyan,*** *we will look at how computers use software.*

ASSEMBLY LANGUAGE PROGRAMMING

PROGRAMMING IN 6502 MACHINE LANGUAGE

by John R. Fachini

This is the first installment in a series of articles concerning machine language programming in the Kyan Pascal environment. We'll start out by using simple examples to clarify the 6502/Kyan Pascal stack interface, 6502 instructions, addressing modes, programming techniques, and allow those people who are just starting to learn machine language to catch up between now and the next issue of the newsletter.

If you are debating whether or not to learn machine code, please do. You'll be amazed at how simple it is, how quickly you catch on, and how much more power you will be able to add to your programs with very little effort.

Now that you've decided to expand your programming knowledge to include 6502 machine code, you need to find a good book which will fill in the most elementary aspects of machine language programming and act as a companion to this series of articles. I suggest Programming the 6502 by

Rodney Zaks, published by Sybex. This suggestion is based on personal experience (I'm not on commission from Sybex).

The actual progression of these articles is not definite; a lot of the information covered in future columns will be determined by you -- the readers. We'll do our best to cover the most requested topics. Just put them down on paper and let Kyan know.

Onto to the business at hand...

The Apple // Series of computers is run by a 6502 microprocessor. The microprocessor controls all displays, devices, and memory usage by following the instructions it finds as it executes a program. Since the microprocessor is very fast (it's program instructions take between 2 and 6 thousandths of a second each to execute, depending on the instruction), the Apple runs programs written in machine code much faster than any other language. In fact, Kyan Pascal is the fastest Pascal available for Apple computers because the Kyan Compiler/Assembler converts your Pascal program into machine language and the 6502 can then execute the instructions directly.

Memory is made up of bytes. A byte consists of 8 bits. A byte can also be broken into 2 nibbles (4 bits per nibble). The following diagram illustrates this structure:

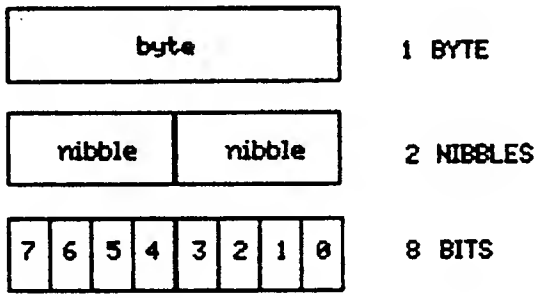


Figure 1.1

Since bytes are made up of binary digits -- bits -- and bits are "logical" digits (value 0 or 1), it is easy to calculate the value range of a single byte. There are 8, base-2 "columns", so the individual bits have values:

bit:	7	6	5	4	3	2	1	0
value:	128	64	32	16	8	4	2	1

Notice the "leftmost" bit has the highest value, i.e. 2 to the 7th power. Moving to the right, the value of the individual bits decreases. This terminology may seem odd at first, but get used to it because the concept of "least significant' and "most significant" will become important in the very near future.

Adding up the values listed results in the decimal number 255. This is the maximum value a single byte can have. Decimal 255 is obviously too small a value to be of use in many cases, so the concept of a 2 byte value (or "word") becomes important.

A word consists of 16 bits (2 bytes). The values in the Most Significant Byte of a word, bit by bit, are:

bit:	15	14	13	12	11	10	9	8
value:	32768	16384	8192	4096	2048	1024	512	256

Adding up all of the individual bit values in a word provides us with a much bigger number to work with: 65535.

The byte containing word bits 15 thru 8 is referred to as the"Most Significant Byte", following the same logic used in naming the most significant bit when working with a single byte.

The microprocessor uses "registers" in which data can be loaded, manipulated and stored again. The Accumulator is a one byte register in which all math functions (addition, subtraction, and bit shifting) must take piace. The other two registers, X and Y, are used mainly for "indexing" and temporary storage of one-byte data.

We're moving pretty fast for the beginners out there (but that's why your supposed to buy a book, right?) At this point I suggest reading the following sections in your 6502 machine-language book:

- o REGISTERS
- o SIGNED NUMBERS
- o BCD (Binary Coded Decimal) NOTATION
- o 16 BIT ADDITION and SUBTRACTION
- o OVERFLOWS and CARRIES

With what you know now, you can still use the demonstration functions which follow. Just be sure to read the above topics before you receive the next issue of the newsletter. I'll briefly summarize the above topics in the next issue (honest). Be sure to read the remainder of the column - I promise you'll learn something new, and you'll need the information later.

For the benefit of the more experienced programmers reading this column, I'll begin this section by explaining the Pascal - Stack interface. If you are using version 1.2 or earlier of the Kyan Pascal compiler/assembler, disregard this information (but get a copy of Version 2.0 - it's worth the investment). If you're just starting out, read this section and save it for reference later on. We'll be reviewing the stack format fairly often so that everyone understands the listings as they are presented.

The Kyan Pascal Compiler/Assembler version 2.0 marks a great step forward in the Kyan Pascal compiler's evolution. The new compiler generates more compact code, offers direct access to a powerful assembler, ISO (International Standards Organization) compatibility -- making it compatible

with most textbooks and Pascal code from other machines -- and an interface into a new, Unix-like operating environment called KIX™.

In order to completely understand the structure of the Kyan Pascal stack structure and environment, it is necessary to first examine the algorithm involved in variable allocation and different types of storage formats.

As the compiler encounters variable declarations, it must allocate memory in which to store variables during the execution of the Pascal program. Variables are allocated memory based on their types. For the time being, we'll concern ourselves with INTEGERs, CHARs, and BOOLEANs. (The other types will be discussed in the next column).

INTEGERs get 2 bytes of storage, providing INTEGER variables a range value of -32768 to 32767 (or 0 to 65535 unsigned). INTEGERs are always stored so that the Least Significant Byte (LSB) is stored one byte "lower" than the Most Significant Byte (MSB). Lower means that the LSB of an INTEGER value is stored in a memory location one byte less than the location in which the INTEGER'S MSB is saved. The example at the end of this section will clarify further. This convention is used for all 2 bytes values.

CHARs and BOOLEANs are 1 byte variables. The byte which represents a CHAR is the ASCII code corresponding to that character. BOOLEANs are either zero or non-zero bytes, zero indicating a FALSE boolean value.

Each procedure and function in the Pascal program must know exactly how the variables available to it are stored. Thus the "stack" convention. Through the stack, a procedure or function can calculate the position of any variable declared locally (a similar task is performed for references to Global variables). We are concemed at this point only with the local stack area.

The calculation (or miscalculation) of variable positions on the local stack by programmers is the primary cause of bugs in assembly language routines interfaced to Kyan Pascal programs. Most of these bugs are easily avoided if a few steps are taken when developing 6502 routines for your programs.

Guidelines for Writing Assembly Language Routines

1. Upon entry to your routine, be sure to save the value of the X register and restore it before returning to Pascal.

2. Carefully map out the local stack before writing any code which references values stored there.

3. Don't try to return to Pascal with an RTS, JSR or JMP. Always use the # convention to allow the compiler to generate the correct return machine code for you. If your code marks the end of a procedure or function, be sure to end it with the # in column 1, followed by the "END;" Pascal marker. The "END;" generates the return-from-procedure code which the system must have in order to behave properly.

4. Use memory in zero page locations _T thru _T+15 only. Do not store data in memory from $300 to $340 or on Bank 2 of MotherBoard RAM! If you do so, KIX will not function properly. Also, the Version 2.0 Assembler uses labels with an underscore (_) identifier. You should avoid using labels which begin with an underscore in your own programs.

5. Any values you save while executing a 6502 routine which you want to keep safe from the rest of the system must be saved in a data area you allocate in your assembly language routine. Zero page _T thru _T+15 is not safe after Pascal regains control of the program.

6. The offset into the local stack for variable storage is now 5 bytes (the stack offset in older versions was 3 bytes).

In order to reference the local variable stack, Pascal provides a pointer to the top of the local stack from which to reference the values stored there. The assembler pre-defines this location as _SP. This "pointer" is actually a pair of zero-page locations in which the address of the top-of-stack byte is stored. Indexing can be done easily using the Y register (don't worry if you don't understand -- an explanation is on the way).

It's actually much easier than it looks. The most complex part of the above guidelines is the mapping of the local variable allocation. The following example should clarify it.

Assume you have declared:

FUNCTION EXAMPLE(firstparm:INTEGER; secondparm:BOOLEAN; lastparm:CHAR): INTEGER;

VAR firstlocal,secondlocal,lastlocal: INTEGER;

The local stack for FUNCTION EXAMPLE would then be:

Offset from _SP	Stack	Description
0	[ ]	Lexical level
1	[ ]	LSB of old _SP
2	[ ]	MSB of old _SP
3	[ ]	LSB of caller
4	[ ]	MSB of caller
5	[ ]	LSB of LASTLOCAL
6	[ ]	MSB of LASTLOCAL
7	[ ]	LSB of SECONDLOCAL
8	[ ]	MSB of SECONDLOCAL
9	[ ]	LSB of FIRSTLOCAL
10	[ ]	MSB of FIRSTLOCAL
11	[ ]	LSB of EXAMPLE return value
12	[ ]	MSB of EXAMPLE return value
13	[ ]	CHAR of LASTPARM
14	[ ]	BOOLEAN of SECONDPARM
15	[ ]	LSB of FIRSTPARM
16	[ ]	MSB of FIRSTPARM

Figure 1.2

The variable names are intended to provide a general way of mapping out the stack. The following algorithm can be used to map the stack:

1. Count "down" to the byte at offset 5. This is the first stack location of interest to you.

2. Allocate bytes, moving down thru the stack, for the LAST LOCAL VARIABLE declared by the Pascal procedure/function.

3. Allocate the second-to-last variable, still moving more deeply into the stack.

4. Repeat until you have allocated memory on the stack for the FIRST LOCAL VARIABLE.

5. If the 6502 code is part of a PROCEDURE, skip to step 7.

6. If the 6502 code is part of a FUNCTION, allocate memory according to the FUNCTION type (i.e., if the FUNCTION returns an INTEGER, allocate it 2 bytes, LSB first). Your 6502 code will put the returned value in this memory.

7. Allocate stack space for the LAST PARAMETER in the parameter list in the Procedure/Function header.

8. Allocate stack space, working to the left until you have provided stack memory for the first parameter passed. The stack calculation is complete.

This algorithm may seem somewhat intimidating at first, but practice with it. After a few tries you'll see how simple the procedure makes drawing out the stack maps. For practice, try mapping the FUNCTION EXAMPLE above, then comparing it with the illustration.

After all this work, we'll exit with a pair of short routines. The first one returns the number of ProDOS devices attached to your computer. The second returns TRUE if paddle button 0 or the open-apple key is pressed.

```
FUNCTION DeviceCount: INTEGER;
{ Return number of active devices }
BEGIN
#A
        LDY   $BF31
        INY
        TYA
        LDY   #5
        STA   (_SP),Y; function value only at top of stack
        INY
        LDA   #0              ;MSB of integer  is zero
        STA   (_SP),Y
#
END;
```

```
FUNCTION OpenApple:BOOLEAN;
{return TRUE if paddle button 0 or open-apple pressed}
BEGIN
#A
        CLC
        LDA   $C061          ;status location
        AND   #$80
        ROL
        ROL
        LDY   #5
        STA   (_SP),Y
#
END;
```

That's all for now. If you're just beginning with assembly language, get to work on those books! If you're more experienced, be patient. The next column will be a little more interesting, we'll cover passing pointers, arrays, and call-by-reference parameters.

Remember ... you will determine the direction of this column. Please write and let us know what topics you would like to see us cover.

PASCAL PROGRAMMING

We would like to pass on a letter we received from Avram Rudy Vener, an **Update ... Kyan** subscriber. Mr. Vener's letter covered a number of different topics. The first of which relates to our last issue where we discussed chaining in Kyan Pascal. He states:

"My problem was that I needed large chunks of code to perform a variety of record and file manipulations. This code could be written easily enough, but I could not compile all of it at one time since I invariably (no pun) overflowed the symbol table. The chaining command as it now exists only works in one direction and passes variables as described in the manual. I needed the different modules not only to share variables but also to return to specific modules upon the completion of their tasks, depending on user input as well as from which previous module the current one was chained from.

"The solution to these problems is relatively straightfoward. First, declare all TYPES in a single file. This file can then be included in all your separately compiled modules so that all TYPES are ensured to be identical. Second, define a set of SYSTEM variables. That is, variables which define the system you are using. For example in my current application I have designed what can be considered a file manager that permits certain users limited access to specific catagories of files. I therefore defined a record type called Currentuser which contains ALL the pertinent information about who is using the system at the moment. Currentuser is the first variable declared in the main program's VAR section in every module. Other system variables are also declared in specific order including one special variable called LINK.

"LINK is a variable of type char. It is used to determine the module which chained the current module. For example, one of my modules is a simple line editor. It can be called by any of seven other modules. In each of those seven modules I have few lines of code which look like this:

```
Link:= 'E';
chain('editor.o');
```

"The editor.o module then permits the creation of a text file called 'temp'. Once the file has been created and editing is complete the editor must then chain to the next file in the system. That is where LINK comes in. In the editor module is a case structure as follows:

```
case LINK of
        'E': chain('e.mail.o');
        'F': chain('feedback.o');
        'B': chain('bulletin.o');
          ...
          ...
        end; (*case*)
```

"Boolean switches and flags in system variables can be set to determine at what condition a module is being entered. For example, one variable can be called FirstTime and can be set True from your startup module. Once the module is entered it can be set false. This allows certain actions such as printing a menu the first time a module is entered but only the prompt line on subsequent chains to that module.

On another topic, one of the nicest things about Kyan Pascal is its ability to let the programmer use inline assembly code. This is a truly marvelous feature which can facilitate certain tasks that would be very cumbersome in Pascal. For example in one input routine, it would be desirable to mask out the eighth bit in a data char. This is trivial using the inline assembler:

```
Procedure MyRead;
var
  c: char;
begin           (* used to read from serial port, assumes IN#2 *)
  read(c);
#a
        LDY #3          ; Offset to C variable
        LDA (SP),Y      ; Put C data into accumulator
        AND #$7F        ; Mask out eighth bit
        STA (SP),Y      ; Put it back into the C variable
#
.... (* continue with Pascal stuff *)
end;
```

You can switch back and forth between Pascal and assembly many times within a single procedure. For example:

```
Procedure Foo;
begin
 for i := 1 to MaxLength do
  begin
  ...
  ...
  (* check for a serial data *)
#a
        LDA   $COA8          ; Get status flags
        AND   #8             ; Check bit number four
        BEQ   NOINP          ; No input data yet
#
        (* set indata flag onlyif there IS data *)
 indata := true;
#a
        NOINP EQU    *
#
 (* continue with Pascal stuff *)
...
end;
```

What happens is that the line: *indata := true;* is skipped if the AND test in the machine code results in a zero. The trick in using inline assembly is knowing when to use it and how much. There are no hard and fast rules, but in general I only use inline assembly to perform data manipulation on the byte or bit level. I let Pascal handle floating point, but I might perform a lower case to upper case conversion using inline code.

Another place where inline assembly comes in useful is with the MLI. Kyan provided a PR routine. I also needed an INP routine. It is listed below.

```
Procedure INP(slot:integer);
begin
#a
  STX T              ; Save the x Register, Kyan needs it later
  LDY #3             ; Offset to SLOT variable
  LDA (SP),Y         ; put slot number (LSB) into a jump to set input routine
  JSR $FE8B          ;
  LDX T              ; replace the X variable for Kyan
#
end;
```

LETTERS

"I became very tired of using the Filer to delete my old compilations and free disk space, so I wrote a routine that could do it from the prompt (PROGRAM KILL). Note that I handle my error codes codes by simply printing out a generic message. Kyan's error routine drops me into the monitor, and, since I never remember the exact spelling of my files, I spent a lot of time there!
... AVRAM RUDY VENER, Weston Connecticutt.
[Editor's Note: Sorry about that! Please try version 2.0 and KIX].

```
PROGRAM KILL (input,output);
const
  maxpath = 65;
type
  pathtype = array[1..maxpath] of char;
var
  c : char;
  path : pathtype;
  i : integer;

procedure destroy (var namebuf:pathtype);
begin
#a
DEST          EQU   *
              LDA   #01
              STA   P
              LDY   #3
              LDA   (SP),Y
              STA   P+1
              LDY   #4
              LDA   (SP),Y
              STA   P+2
              JSR   MLI
              DB    $C1
              DB    >P
              DB    <P
              BCC   DSTEND
#
  writeln(' Error - Kill aborted ');
#a
DSTEND        EQU   *
#
end; (* procedure destroy *)

begin
 reset(input);
 rewrite(output);
 i := 1;
 writeln;
 write(' Pathname? ');
 while (not eoln) do
        begin
         i := i + 1;
         read(c);
         path[i]:= c
        end;   (* while not *)
 path[1] := chr(i-1);
 destroy(path);
end.
```

Mr. James Luther of Kansas City, Missouri has contributed the following assembly language routines to the readers of **Update ... Kyan**. The routines all use the PEEK and POKE routines included in Volume 1, Number 1 of this newsletter.

This function reads game controller buttons 0 thru 3 and returns TRUE if a button is pushed.

```
Function Button (Select:Integer): Boolean;
  Begin
   If (Select >= 0) and (Select <= 3)
   Then
        Case Select of
        0 : (* Button 0 *)
          If Peek (-16287) > 127
          Then Button := True
          Else Button := False;
        1 : (* Button 1 *)
          If Peek (-16286) > 127
          Then Button := True
          Else Button := False;
        2 : (* Button 2 *)
          If Peek (-16285) > 127
          Then Button := True
          Else Button := False;
        3 : (* Button 3 is the cassette input *)
          If Peek (-16288) > 127
          Then Button := True
          Else Button := False
        End  (* Case *)
    Else  (* Select out of range *)
        Button := False;   (* so make it false *)
End;
```

This function checks to see if a key has been pressed and returns TRUE if it has. You can use a read command to get the character.

```
Function Keypressed : Boolean;
Begin
  If Peek (-16384) > 127
  Then Keypressed := True
  Else Keypressed := False;
End;
```

This procedure plays a musical note (0 through 51) for duration (0 through 255). Note 0 produces a silent tone for the duration specified.

```
Procedure Note (Pitch, Duration : Integer);
Begin
  If (Duration >= 0) and (Duration <= 255) and (Pitch >= 0) and (Pitch <= 51)
  Then
#A
        TXA
        PHA                     ; saves X-register
;
        LDY   #5
        LDA   (SP),Y            ; get pitch
        TAY
        LDA   NOTES,Y
        STA   T
;
        LDY   #3
```

```
        LDA   (SP),Y
        STA   T+1             ; duration at T+1
;
        LDY   #0              ; fast duration count
;
NT1     LDX   T
        CPX   #$FF
        BEQ   NT4             ; branch if silent ($ff)
        BIT   $C030           ; whap the speaker
;
NT2     DEY                   ; decrement fast count
        BNE   NT3             ; if no borrow
        DEC   T+1             ; decrement duration
        BEQ   NT5             ; if finished
;
NT3     DEX                   ; decrement pitch value
        BNE   NT2             ; pitch not done
        BEQ   NT1             ; pitch done, always taken
;
NT4     INX                   ; trap X to $ff
        BEQ   NT2             ; always taken
;
NOTES   DB    255             ; pause
        DB    232,219,207,195,184,174,164,155,146,138,130,123 ; oct#1
        DB    116, 110, 103,  98,  92,  87,  82, 78, 73, 69, 65, 61 ;oct#2
        DB     58,  55,  52,  49,  46,  44,  41, 39, 37, 35, 33, 31 ;oct#3
        DB     29,  27,  26,  24,  23,  22,  21, 19, 18, 17, 16, 15 ;oct#4
        DB     14,  13                                              ; oct#5
;
NT5     PLA
        TAX
#
End;
```

This function reads the game controller 0 to 3 and returns an integer 0 to 255.

```
Function Paddle (Select : Integer) : Integer;
Begin
  Paddle := 0;  (* make sure high byte is zero *)
  If (Select >= 0) and (Select <= 3)
  Then
#A
        TXA
        PHA                   ; save X-register
        LDY   #5
        LDA   (SP),Y
        TAX                   ; X <- select
        JSR   $FB1E           ; read the paddle X
        TYA
        LDY   #3
        STA   (SP),Y          ; save paddle
        INY
        LDA   #0
        STA   (SP),Y
        PLA
        TAX                   ; restore X-register
#
  Else Paddle := 255;   (* out of range *)
End;
```

UPDATE ... KYAN

Kyan Software Inc.
1850 Union Street #183
San Francisco, CA 94123

May/June Issue
Volume 1, Number 4

Apple Edition

(Editor: Sonja Newell)

WHAT'S NEW?

Telecommunications and Kyan

For those of you who have accounts on CompuServe, you should know that Kyan has an account, too. Our PPN is 73225,450 and we can be found in a few forums on CIS (Sorry, but we don't loiter around CB!). The most effective way of communicating with Kyan is via E-mail, not SIG messages; although forum messages are a fine way of asking other users for help or offering suggestions. A few routines have been uploaded to CompuServe forums by our users, and we would like to see more. We also may start uploading to the Data Libraries ourselves. If you have a CIS account and you would like to communicate with other Kyan users on CompuServe, send us E-mail and/or post messages in the forums. We will make your PPN identification number available to other Kyan users and vice versa.

Users of MCI Mail can also send us E-mail to mailbox : 298-0892. Western Union EasyLink subscribers can send telexes to: 989113 KYAN SFO

E-mail is an effective way of getting technical support from Kyan. Short listings and problem descriptions can be E-mailed and may get a reply very quickly. We would like to reiterate our technical support policy: We can help you with problems or possible bugs with any of Kyan's products, but we cannot write or debug your programs for you.

We would like to know how many of the Kyan users/programmers out there are using modems. If you have a modem, please indicate so on the reader survey within this issue of **Update...Kyan.**

In addition to our CIS and MCI accounts, we are pondering the possibilty of a Kyan BBS (Bulletin Board System). If you do not yet have a modem, this could be a good reason to get one! The BBS would most likely be up sometime after we close the office here in San Francisco(after 5:00pm PDT). Messages could be posted concerning technical support and programs, and newsletter routines could be up- and down-loaded. It is also possible for us to put updates on a BBS so users can update their software without having to send their disks to us. If you would like to use a Kyan BBS please let us know. We are open to suggestions and would very much like to hear your thoughts about this.

Software Submission Program

We would like to encourage readers to submit programs for publication in **Update...Kyan**. As incentive, we will pay authors of Pascal or Assembly Language routines a reward of $50.00 for each routine published. To be eligible, the routines must be creative and original. Also, the source code should be fully commented. We are open to any and all sorts of programs; the following may give you some ideas.

- o Music/Sound routines
- o Graphics/Text and Graphics routines
- o Input/Output driver routines for accessing slots
- o Unique data structures

Please try to keep your programs at a reasonable length. All routines accepted for publication in **Update...Kyan** will become public domain software without any limitations regarding duplication or distribution. Routines not accepted for publication will remain the exclusive property of the author.

Kyan Pascal Update

As we promised, we will now look at computer hardware and how it works with the software you write. The following is a continuation of the last issue's article and is an excerpt from **Oh! Pascal!** by Doug Cooper and Michael Clancy (second edition; W. W. Morton, New York, NY 1985) Once again we'd like to emphasize what a fine text it has for learning Pascal. Not only is it easy to follow but the authors don't get bogged down in dry text.

****

"It may be somewhat disconcerting to find that even though this text is devoted to the subject of computer programming, computers themselves are almost never mentioned. In practice, programmers don't have to know much about the internal workings of a computer, any more than typists need to understand the mechanical underpinnings of an electric typewriter. But rather than succumb to this appeal for ignorance, let's get an overview of what goes on behind the keyboard.

"By now everybody must know that computers are systems with two sides: hardware and software. Neither is of much use without the other and the programs we'll learn to write depend on both." ... "Hardware components include the computer, as well as machines that are connected to it: terminals, printers, secondary memory, and so on. Software components are include applications programs and the operating system.

"It's difficult to separate hardware and software, since even the simplest tasks rely on both. We need hardware to enter data, but we need software to make sure that the computer can communicate with a terminal. We need hardware to print a hard copy of stored information, but we must have software to transfer the letters, one at a time, from the computer to the printer. We need hardware to actually compute figures, but we need software to prepare our figures for computation. Let's look at each part of the system in turn.

"A computer's hardware can be divided into three groups of electronics. The *CPU*, or central processing unit, is the heart of every computer. It runs programs, performs calculations, and manages the operation of the computer's other parts. *Memory* is the second essential component. It stores almost all information the computer uses, from the data needed for program steps that will take place a microsecond hence, to database information that might not be used for years. Finally, *I/O*, or input/output, devices are necessary for communication between a computer and its human users, or other electronic devices.

"The CPU provides what people think of as the brains of the computer. The CPU has several parts that work together closely. The *execution registers* hold program instructions while they are being executed, or carried out. Typically, only the current instruction will be held, which means that from the viewpoint of the execution registers, there is no difference between big programs and little ones, or hard programs and easy ones. The execution registers are also in charge of keeping track of current information - values that are in the process of being changed, currently active locations in memory, and the like.

"Determining the effect of each program step is largely the province of the *ALU*, or arithmetic and logic unit. The ALU is the CPU's decision-making unit. Its primary task is to make small comparisions (are these values equal? which is greater?) that, when taken in great number, seem to be reasoned decisions. To support its decision-making capability, the ALU also carries out elementary arithmetic operations like addition and subtraction. Again, each small step may be insignificant in itself, but they can eventually add up to give the computer the illusion of sophisticated mathematical ability.

"The execution registers and ALU work hand-in-hand, with the registers posing the questions and the ALU supplying the answers. The *CU*, or control unit, keeps them in touch with each other , and also with the rest of the computer (the memory and input/output devices described below). In effect, the control unit serves as the machine's traffic controller.

"Together, the execution registers, arithmetic and logic unit, and control unit largely determine how 'powerful' a given computer appears to be. One basic difference between computers is the speed at which the control unit is able to transfer information between the ALU and registers. A second is the amount of time required for the ALU to actually make a computation. A third is the amount of overlap that can occur - expensive systems will have additional execution registers and ALU's so that work can begin on a new program step before the old one is completely finished. A final power-enhancement mechanism is only

found in the most advanced computers. They let operations take place in parallel - several program steps are carried out simultaneously.

"The CPU stands poised to carry out any single step of a computer program. To do useful work, though, the CPU must work with a computer's other components. *Memory* is the most important. It's needed to store programs that govern a computer's operation, along with data needed when programs are run, partial results that are derived and must be maintained during the course of a program's operation, and any final results that are saved for perusal later.

"Memory is usually divided into two varieties - *main* or *primary* memory and *secondary* memory. Main memory comes with the computer; it's built in, but can usually be increased by purchase of additional 'memory boards' or 'memory chips'. Simply stated, main memory stores running programs and the information they currently use. It is directly in the service of the CPU, which means that the CPU (which is very fast) can go to and from main memory to get new program steps, and to store or retrieve data. Main memory is usually one of the more expensive hardware components, so computers will typically have just enough to meet the requirements of the largest programs they're liable to encounter.

"Secondary memory holds programs and information that are not currently being used. Most people have seen the floppy disks that serve as secondary memory for personal computers. Typically, each floppy disk will store the instructions for (and have room for the results of) a single program. The computer user has to physically insert the disk in order to transfer its contents to main memory, and run the program it holds,

"Secondary memory for bigger, shared, computers is generally made from large, rigid disks that are permanently mounted alongside the computer. Each user has a share of this common secondary storage, and doesn't have to divide her programs, or results, amongst floppy disks. Multi-user computers automatically carry out any transfers between secondary and main memory for the benefit of the computer user.

"I/O (for input/output) devices form the third major component of computer hardware. I/O devices are specialized machines for communication between computer and people, or other computers. Without I/O devices, we'd have no way to store programs, or to supply data when they ran, or to receive results when they were through. An output device, like a printer, can be used to get the results of running a program, or to inspect data stored in another computer component - say, the main memory. Input devices serve a complementary purpose; we use them to supply the CPU (or memory, through the CPU) with new information.

"Network connections are a relatively new addition to the I/O family. They allow extremely high-speed communication between different computer systems. Networks are most commonly used to let computer systems share access to certain components, like printers or terminals. However, experimental networks can let the CPU on one computer interact with the memory of an entirely different machine, or give two different users the illlusion of being on the same system.

"How do we compare the hardware of two different computers? The *speed* of a computer's CPU is a convenient reference because, regardless of the computer's price or its programmer's ability, its programs are almost invariably still executed one step at a time. Since a computer's actual computing is completed in the give and take of its control unit, execution registers, and ALU, the amount of time a single step takes provides a reasonable basis of comparison for two computers that will be expected to run the same sort of programs.

"We can also compare two computers by their *size*. This measurement is usually concerned with the amount of primary memory the computer has, or is capable of having. This can be important, because some programs may require very large amounts of primary memory to run effectively, or to run at all. Finally, we can compare computers on the basis of peripherals. These include secondary memory, as well as input and output devices.

"All the measures described above have their uses. However, we generally find that criteria like CPU speed, memory size, and peripherals alone are best used for comparisons of smaller computers intended for personal use. For larger computers that will probably be shared between a number of users, though, these measurements are often too simple. The prospective purchaser must try to determine just how harmoniously the separate hardware components work together. Ultimately, this will often turn out to depend on our next topic - software.

"Computer software falls into two major categories: applications programs, and operating systems. *Applications programs* are specific; each program does one particular job when (and only when) a computer user requests it. The *operating system*, in contrast, is a general piece of software. It runs continuously to coordinate the operation of the computer's hardware and software resources. In effect, computer users interact directly with applications programs, while the operating system is more closely associated with actual hardware.

"When we think about software we usually imagine applications programs - programs that do some specific task. Some typical applications programs are:

- Programs for word processing and text editing.
- Programs for game playing.
- Programs that handle accounting or arrange spreadsheets.
- Programs that help with instruction.
- Programs that prepare programs to run on the computer.

"A look at any computer magazine will provide dozens of additional examples.

"By itself, though, computer hardware isn't sophisticated enough to run applications programs. Even though the control unit coordinates hardware operation in a low-level way, there must still be a connection between applications programs and the actual computer hardware. Why? Well, a typical applications program needs more that just raw computational power. It will undoubtedly require the services of different input and output devices. It may be stored with other programs in the computer's memory, and have to be retrieved before it can be run. It might even need hardware that's currently being used by another programmer.

"This is where operating system software comes in. The operating system is a very large program that controls and coordinates the operation of computer hardware for the benefit of individual users and their programs. Now, if a computer just ran one program and had no peripheral equipment - like the microprocessor found in a toaster or carburetor - it wouldn't really need an operating system. The point of an operating system is to create an environment in which different applications programs, which use the computer system in a variety of ways, can be run. Since the operating system is so necessary, it is almost invariably supplied with the computer directly from the hardware manufacturer.

"What are some of the practical problems that an operating system deals with? For one thing, the operating system *controls computer access*. When you enter a password to log onto a computer, you do so at the operating system's request. The operating system organizes users; it, rather than you, is responsible for keeping stored programs and information that belongs to dozens or hundreds of computer users separate and retrievable.

"The operating system also *allocates independent resources*. Suppose that several computer users want to use a single printer. Each user might independently request that some stored information be printed; it is the operating system's job to form a waiting list so that requests can automatically be handled in turn.

"Not all resources are independent, so the operating system must also *schedule shared resources*. Although a printer can only handle one job at a time, faster resources (like the CPU) can be shared between users. The result of resource sharing is that each system user gets the illusion that she is working on her own personal computer. Behind the scenes, though, the operating system must work furiously to ensure that individual users' programs don't get mixed up as each takes its turn at using the CPU and main memory.

"From our point of view as computer programmers, the operating system's main job is managing the programming environment. It supplies the tools we need to write programs, then arranges for our programs to run. As a result, we don't have to worry about the myriad details that are involved in computer operation.

****

Thank you Professors Cooper and Clancy! The preceding text is an excerpt from their textbook **"Oh! Pascal!"**.

ASSEMBLY LANGUAGE PROGRAMMING

PROGRAMMING IN 6502 MACHINE LANGUAGE

by John Fachini

With the release of Kyan Pascal version 2.0 and the KIX environment, a few 'loose ends' have cropped up over the past couple of months. So instead of proceeding with the usual beginners column, I've decided to use the column this month to provide everyone with some requested routines, clarifications, and suggestions.

WRITING FUNCTIONS: The first point has to do with writing FUNCTIONs using Kyan Pascal version 2.0. Since 2.0 conforms to the International Standards Organization of syntax and implementation, the contents of a FUNCTION written completely in assembly code must be changed from 1.2.

The first difference is the local stack interface (remember from last month that the stack offset starts at 5 bytes deep instead of 3 as in Kyan Pascal version 1.2). Due to the ISO compatibility issue, a second difference has surfaced. ISO requires that the function value be explicitly assigned a return value; assigning the function identifier a value implicitly from assembly language does not satisfy the ISO requirement. This may seem to be a nuisance but, as usual, assembly language programmers can easily turn a nuisance into an advantage.

Two ways to tackle the function limitation are what we'll call the 'preferred method' and the 'lazy method'. First, the lazy method. Consider the following function:

```
FUNCTION LAZY_EXAMPLE: INTEGER
BEGIN
        LAZY_EXAMPLE:= 0;  { used to satisfy ISO requirement }
#A
.............
etc.            ;code in function body assigns actual return value
.............
END;
```

There is really nothing wrong with writing functions this way. But, why waste an assignment? There's another solution which makes debugging easier and doesn't waste an assignment operation. Look:

```
FUNCTION BETTER_EXAMPLE: INTEGER;
VAR
        RESULT: INTEGER;
BEGIN
#A
.............
etc.            ; store function value in local variable
RESULT
.............
#
        BETTER_EXAMPLE:= RESULT
END;
```

Note the two major differences:

1. ISO is satisfied with a non-wasted assignment
2. The resulting function value can be checked from Pascal

To keep the 'better way' simple, make sure that the local variable you use to temporarily store the function value is the last local variable declared. This way, you know the function value destination always has it's lowest byte at stack offset 5.

Recall the two functions listed in the newsletter? Here's the 'working versions' of each:

```
FUNCTION DEVICECOUNT: INTEGER;
{ Return number of active devices }
```

```
VAR
        RESULT: INTEGER;
BEGIN
#A
        LDY     $BF31
        INY
        TYA
        LDY #5
        STA     (_SP),Y         ;'result' value @ tos
        INY
        LDA     #0              ;MSB of integer is zero
        STA     (_SP),Y
#
        DEVICECOUNT:= RESULT      {satisfy ISO }
END;

FUNCTION OPENAPPLE: BOOLEAN;
{Return TRUE if paddle button 0 or open-apple pressed }
VAR
        RESULT: BOOLEAN;
BEGIN
#A
        CLC
        LDA     $C061           ;Status location
        AND     #$80
        ROL
        ROL                     ;Puts a zero in Accumulator if off, 1 if on
        LDY     #5              ;'Result' gets function result first
        STA     (_SP),Y
#
                OPENAPPLE:= RESULT        {ISO strikes again!}
END;
```

DISABLING LOWER CASE OUTPUT: We've received a few requests from Apple][+ users with 80 column cards as to how to disable lower case output. First, an explanation, then a solution.

The KIX environment determines the case of the characters it outputs to the screen based on the presence of an 80 column card in the system. If an 80 column card is found, lower case characters are used; otherwise, upper case only is used.

The problem for Apple][+ users with 80 column cards occurs when they go to 40 column mode while in the KIX environment. Since KIX still sees an 80 column card, it continues output in lower case. The solution is the following Pascal program, you should run this program when you see the KIX prompt.

```
PROGRAM DISABLE_LOWERCASE;
BEGIN
#A
        LDA     $BF98  ;Load the Machine I.D. Byte
        AND     #2     ;Tell system no 80 column card
        STA     $BF98  ;And save change
#
END.
```

Memory location $BF98 is the ProDOS Machine Identification byte. When the system is first booted, ProDOS looks at all the peripherals in each slot of the computer and identifies each by matching certain 'hardware identification bytes'. When ProDOS finds an 80 column card it makes the second bit in the Identification byte a '1' ('on'). The previous program makes that bit a 0 ('off'). this change will work until you reboot your system or re-execute the 'ProDOS' system file.

The entire topic of ProDOS provides great potential for future columns. If you have any questions about ProDOS or your Apple in general please send them in, and I'll do my best tio answer them. To be honest, I

was a bit underwhelmed by the reaction to the last column. If you don't write and tell me what you want to see in this column, we'll probably end up going in circles and jumping aroung to all different topics.

CHAINING SYSTEM FILES: We have had many requests for a routine which allows programmers to CHAIN programs when not in the KIX environment (on a stand-alone disk, for example). the current version of CHAIN will not support this because CHAINing in version 2.0 is done via the KIX operating environment. Since KIX is not present on the stand-alone disk, the call to CHAIN causes the system to die. The rule is "never mention a problem without a solution", so here's the solution, another CHAIN procedure:

```
PROCEDURE CHAINPROGRAM(VAR PROGRAMNAME:PATHSTRING);
```

The format of the call is pretty obvious, but the type "PATHSTRING" may not be. (Anybody with a System Utilities Toolkit should recognize PathString!). A PATHSTRING is:

```
TYPE
        PATHSTRING = ARRAY[1..65] OF CHAR;
```

Recall from your Kyan Pascal 2.0 manual that a ProDOS pathname cannot be longer that 64 characters. In order to use a pathname from assembly language, the letters in the pathname must be moved 'back' one byte so that the first byte in the aray is the number of characters in the pathname. Why must this shift be done? It's because of the ProDOS MLI:

The ProDOS MLI (Machine Language Interface) provides programmers with a very easy interface to ProDOS file operations. Each call has the same format:

```
JSR     _MLI
DB      opcode_byte
DW      location_of_parameter_list
```

The _MLI is at location $BF00 and is predefined in the STDLIB.S compiler macros file. The opcode_byte corresponds to the one-byte operation code. This byte tells the MLI exactly what you want to do. The location_of_parameter_list field is the address (lo byte, hi byte) of the list of parameters that ProDOS will need in order to perform the operation you have requested.

We'll go into more detail about the MLI and ProDOS in a future entry in the newsletter. For now, use the following procedure and, if you don't understand it, carefully read the remarks a few times. If you are even more curious about ProDOS and the MLI, get a copy of "BENEATH APPLE PRODOS" from Quality Software.

```
PROCEDURE CHAINPROGRAM(VAR PROGRAMNAME: PATHSTRING);
{Chain program "pathname" to current program. Note that no error checking takes place
in this code so care should be taken when calling it.}
BEGIN
#A
        LDY     #5              ;Stack address of VAR parameter
        JSR     MAKEPATH        ;Make string into pathname
        STX     CHPINFO+1       ;Address (LSB) of pathname
        STX     CHPOPEN+1
        STY     CHPINFO+2
        STY     CHPOPEN+2
        JSR     _MLI
        DB      $C4             ;Get File Info
        DW      CHPINFO         ;Parameter list
;By using the load address instead of file type, you can use this routine to chain pascal
;programs to assembly language programs and vice versa. The only exception to this is
;SYSfiles which are loaded at $2000 and self relocating to either $800 or $4000
        LDA     CHPINFO+4       ;program type
        CMP     #$FF            ;SYStem file?
```

Press RETURN for more; type NO to stop

```
        BNE     CHP1            ;No
```

```
        LDX   #>$2000
        LDY   #<$2000
        JMP   CHP1A          ;Set load address to $2000
CHP1    EQU   *
        LDX   CHPINFO+5      ;Load the porgram at whatever address
        LDY   CHPINFO+6      ;it was saved at
CHP1A EQU   *
        STX   $BEFE
        STY   $BEFF
CHP2    EQU   *              ;Open the file so it can be read
        JSR   _MLI
        DB    $C8            ;Open call
        DW    CHPOPEN
        LDA   CHPOPEN+5      ;Put resulting reference number
        STA   $BEFD          ;at another safe byte
;The last operation this code has to perform is a relocation of the code which will read the
;chained file so that the subsequent close and jump to execute the new code does not
;get overwritten by the incoming file.  Note that we aren't worried about overwritting the
;LIB file ($9000-$BEFF) since each Pascal code segment re-loads the LIB file anyway
        LDX   #0
CHP3    EQU   *
        LDA   CHREL,X
        STA   $BE00,X        ;Safe for use
        INX
        CPX   #200           ;Move up to 200 bytes (plenty extra)
        BNE   CHP3
        JMP   $BE00          :Execute the relocated code
;
;-------------------------------------------------------------------------------------------
; Code to relocate:
CHREL EQU   *
RELCON  EQU $BE00-*          ;Relocation constant
        LDA   $BEFD
        STA   CHPREAD+1
        LDA   $BEFE          ;LSB of starting address
        STA   CHPREAD+2
        LDA   $BEFF
        STA   CHPREAD+3
        JSR   _MLI
        DB    $CA            ;Read request
        DW    CHPREAD
        LDA   #1
        STA   CHPREAD        ;Make read list into close list
        JSR   _MLI
        DB    $CC            ;Close
        DW    CHPREAD
        JMP   ($BEFE)        ;And execute new program via indirect jump
;
CHPREAD  EQU *+RELCON
        DB    4              ;Read call has 4 parameters
        DB    0              ;Reference number
        DW    0              ;Address to load into (set by code above)
        DW    $FFFF          ;Load length (as far as we can get)
        DW    0              ;Actual length read (not needed)
;
;-------------------------------------------------------------------------------------------
;Parameter lists used by main code:
;
CHPINFO  EQU *               ;Get File Info parameter list
        DB    10
        DS    18             ;Rest is set by code or call
;
```

```
CHPOPEN  EQU *
         DB    3
         DW    0              ;Address of pathname (set by code)
         DW    $9000          ;Page aligned 1K
         DB    0              ;ProDOS will put the refnum here
;
;-------------------------------------------------------------------------------
;This routine converts the string whose address is (_SP),Y into a 'pathname': a pathstring
;preceeded by a length byte
;
MAKEPATH   EQU  *
         LDA   (_SP),Y        ;Address of string address
         STA   _T+2
         INY
         LDA   (_SP),Y
         STA   _T+3
         LDY   #64            ;Character counter
MPATH1   EQU *
         LDA   (_T+2),Y       ;Get a  byte from the string
         CMP   #32            ;Is this a blank?
         BNE   MPATH2         ;No:  found count
         DEY
         BPL   MPATH1         ;Keep trying until <0
         INY                  ;Make y=0
         TYA
         STA   (_T+2),Y       ;Makes length of string 0
         JMP   MPATH3
MPATH2   EQU *
         INY                  ;Actual path length
         STY   _T             ;Save for later
         DEY
MPATH4   EQU *
         LDA   (_T+2),Y       ;Get a byte of the string
         INY
         STA   (_T+2),Y       ;And push it 'back' one byte
         DEY
         DEY                  ;Fix Y
         BPL   MPATH4         ;Do all byte 0..length
         INY                  ;So that y = 0
         LDA   _T             ;Recover length byte back
         STA   (_T+2),Y       ;Makes string into pathname
MPATH3   EQU *
         LDX   _T+2
         LDY   _T+3
         RTS
#
END;
```

All that typing should keep you busy for a while, at least. Remember: send me some mail! If you have a question, write it down. You're probably not the only person out there with that certain question and you'll be helping everybody by taking the time to write. Go for it.

Next time we'll go back to easier topics, but still try to keep it interesting for those of you who know all that beginners stuff already. Somebody send me and idea!

Version 2.0 -- Troubleshooting Reports

The following problems have been encountered by users of Version 2.0. If you have encountered other problems, please let us know. Also, please read the Letters and Assembly Language Programming sections of the newsletter for more information about 2.0. (NOTE: Owners of Version 2.0 can obtain updated software at any time by exchanging their original program disks. There is no charge for these updates.)

1. Case Statement Errors.

A little known ISO restriction is that the character assigned to the variable in a Case statement must exist in the set of case test statements. For example:

```
CASE CH OF
'A' : action1;
'B' : action2;
'C' : action3;
'D' : action4
END;
```

If CH is equal to F, a Case Index Error will be generated.

A bug has been uncovered in the Compiler Case statement which also causes a Case Index Error. This bug has been corrected.

2. _UsesHires Error.

An error occured in the final translation of the _UsesHires file which causes programs to crash. This error can be corrected by:

a. Load the _UsesHires file with the Kyan Editor and go to line 248.
b. Change: "U EQU 22" to "I EQU 22".
c. Save the file and quit.

3. Chaining.

The Version 2.0 Chain procedure works fine when you chain files under the KIX operating shell. However, if you want to chain system files on a stand-alone disk, you must use a special routine which is found in the Assembly Language section of this newsletter.

4. KIX Command Bug?

Several single disk drive users have reported a bug in the "CP" command when trying to copy a file. We are happy/sad to report that this is a bug in ProDOS and not in KIX. When a file is opened by ProDOS, the device in which the original file volume is located is removed from the open device list by ProDOS. Thus, when the CP command goes looking for a destination device, none can be found. Apple has been aware of this problem for some time but has not fixed it. Unfortunately, there is nothing Kyan or you can do to make CP work correctly with a single drive system.

5. Assembly Language Listing.

The manual contains an error regarding compiler output when using the -S option. The output file is automatically named P.OUT, not Filename.S as stated in the manual. To save this file, use the file Rename command found in the Filer or KIX and rename P.OUT to Filename.S.

Also, the compiler's assembly language output is actually a string of macros. The expansion of these macros is contained in the text file STDLIB.S on the V2.0 disk. If you want a full listing of P.OUT with the macros expanded, add the compiler directive " MEX ON" (be sure to include the space before MEX and ON) on line 1 of P.OUT and assemble the program using the list option (i.e., AS -l).

Data Structures Using an Open Hashing Function

by Sonja Newell

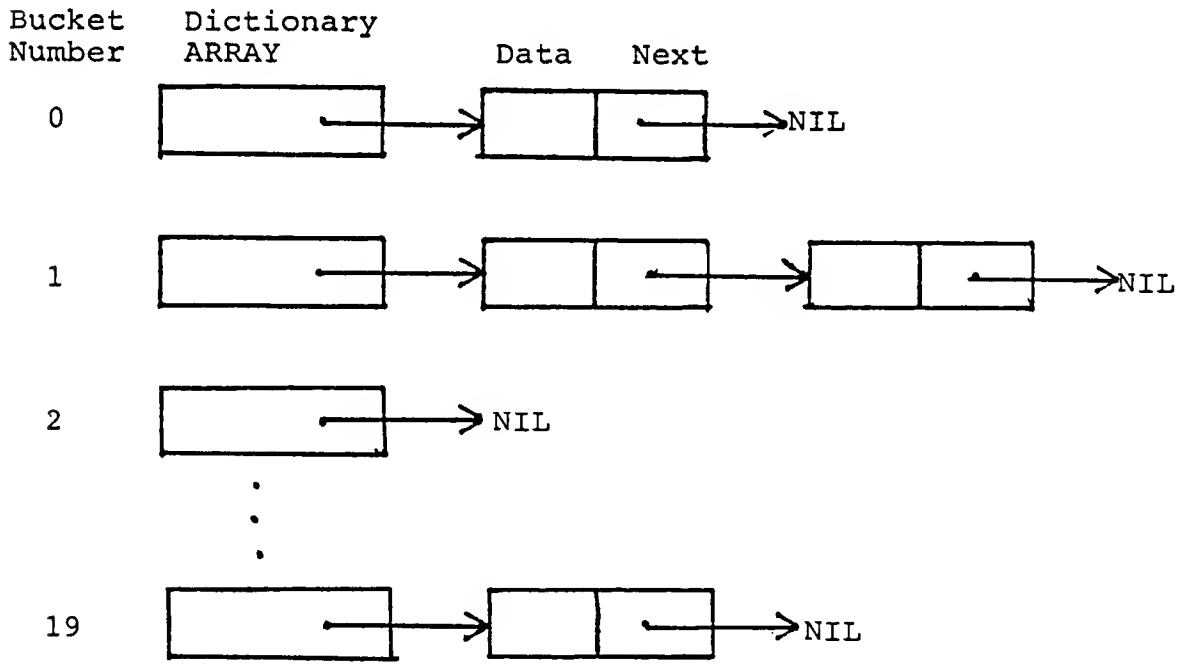
A dictionary, which is used in the following program, is a fundemental data type used in maintaining a set of data. A dictionary can be defined by the basic operations: INSERT, DELETE, MEMBER and MAKENULL. An INSERT procedure is used to put data into the dictionary; the DELETE procedure is simply used to DELETE data from the dictionary; the MEMBER function is used to determine whetеher the data is already in the dictrionary or not; a MAKENULL procedure is used to initialize the data structure.

The following program illustrates the Open Hash Table Data Structure, a widely useful technique for implementing dictionaries. By implementing a hash function, the potential time to search for a record is cut down and therefore your runtime is shorter. Open hashing allows a set to be stored in potentially unlimited space. The RECORDs are stored in an array called a 'bucket table' where the hashed value returned by the function is the bucket number. The bucket number is an index to the dictionary array, therefore by using a bucket number you don't have to search through a whole list of records, but instead through only a designated portion of those records.

First, you need to determine the number of buckets you want to use. Then decide which key field in the record you want to hash, usually a character array. The average time for operations increases rapidly as the number of records exceeds the number of buckets, therefore keep that in mind when choosing the number of buckets in the dictionary. Once you've decided on your data structure you can implement the dictionary procedures and enter your files.

If you want to learn more about hashing (for instance, there is a Closed Hash Table Data Structure), you can refer to most computer science data structure texts. The one I've referred to for this article is "Data Structures and Algorithms" by Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman (Addison-Wesley Publishing Company)

A good practice to get into when using pointers is to draw yourself a picture of your data structure. The picture for this data structure would look like this:



```
PROGRAM Stereo(Input,Output,Albums);

(* Stereo contains the titles of all my albums.  Should I wish to add fields to the RECORD 'Data', such as
'Musicians', 'RecordLabel', 'Instruments', etc. I can easily add them. *)

TYPE
        Buckets = 0..19;
        String = ARRAY[1..20] of CHAR;
        Dataptr = ^Data;
        Data = RECORD
                Title: String;
                Next: Dataptr
                End;

        Dictionary = ARRAY[Buckets] of Dataptr;

(* Note: Do NOT attempt to store a file of RECORDS with a 'Next' pointer field. It won't work *)

VAR
        D: Dictionary;
        Albums: Text;
        F: String;

Function H(X: String): Buckets;
(* Hashing function to determine bucket index *)
VAR
        Sum, J: Integer;
Begin
Sum:= 0;
For J:= 1 to 20 do
        Sum:= Sum + ORD(X[J]);
H:= Sum MOD 20;
End; (* Function H*)

Procedure MakeNull(VAR D:Dictionary);
(* Initialize the dictionary pointer to nil *)
VAR
        I: Integer;
Begin
For I:= 0 to 19 do
        D[I]:= NIL;
End; (* Procedure Makenull *)

Function Member (X: String; D: Dictionary): Boolean;
(* Member checks to see if the record is already in the dictionary*)
VAR
        Current: Dataptr;
Begin
Member:= FALSE;
Current:= D[H(X)];
While Current <> NIL do
        Begin
        If Current^.Title = X then
                Member:= TRUE;
                Current:= Current^.Next;
        End; (* While  *)
End; (* Function Member *)
```

```
Procedure Insert(S:String; VAR D:Dictionary);
(* Inserts a RECORD into a Dictionary *)
VAR
        Bucket: Integer;
        OldHeader: Dataptr;
Begin
If Member(S,D) <> TRUE then
        Begin
        Bucket:= H(S);
        OldHeader:= D[Bucket];
        New(D[Bucket]);
        D[Bucket]^.Title:= S;
        D[Bucket]^.Next:= OldHeader;
        End; (* If Member *)
End; (* Procedure Insert *)

Procedure ReadFile(VAR D:Dictionary);
(* Reads each string from a text file *)
VAR
        A: String;
Begin
Reset(Albums, '/vol/dir/ALS');
While Not EOF(Albums) do
        Begin
        Readln(Albums,A);
        Insert(A,D);
        End; (* While Not loop *)
End; (* Procedure ReadFile *)

Procedure PrintDict(D:Dictionary);
(* Prints all the records in the dictionary *)
VAR
        I: Integer;
        Current: Dataptr;
Begin
For I:= 0 to 19 do
        Begin
        Current:= D[I];
        While (Current <> NIL) do
                Begin
                Writeln(Current^.Title);
                Current:= Current^.Next;
                End; (* While loop *)
        End; (* For I loop *)
End; (* Procedure Printdict *)

Procedure Enter(VAR D: Dictionar);
(* Enter each string into an external text file *)
VAR
        A: String;
        X, Y: Integer;
Begin
Writeln('How many titles?');
Readln('Y);
For X:= 1 to Y do
        Begin
        Writeln('Enter album title');
        Readln(A);
        Insert(A,D);
        End; (* For X loop *)
End; (* Procedure Enter *)
```

```
Procedure Delete(Var D: Dictionary);
VAR
        Current: Dataptr;
        Bucket: Integer;
        S: String;
Begin
Writeln('Which item do you want deleted?');
Readln(S);
If Member(S,D) <> FALSE then
        Begin
        Bucket:= H(S);
        If D[Bucket] <> NIL then
                Begin
                If D[Bucket]^.Title = S then
                        D[Bucket]:= D[Bucket]^.Next
                Else
                        Begin
                        Current:= D[Bucket];
                        While Current^.next <> NIL do
                                If Current^.Title = S then
                                        Current^.Next:= Current^.Next^.Next
                                Else
                                        Current:= Current^.Next;
                        End; (* Else *)
                End; (* If *)
        End; (* If Member *)
End (* Procedure Delete *)

Procedure Writetodisk(D: Dictionary);
VAR
        Bucket: Integer;
Begin
Rewrite(Albums,'/vol/dir/ALS');
For Bucket:= 0 to 19 do
        Begin
        While D[Bucket] <> NIL do
                Begin
                Writeln(Albums, D[Bucket]^.Title);
                 D[Bucket] =  D[Bucket]^.Next;
                End; (* While *)
        End; (* For Bucket *)
End; (* Procedure Writetodisk *)

Begin
Makenull(D);
ReadFile(D);
Writeln ('Type "Finish" when your done with your program');
F:='Flag          ';
While F <> 'Finish          'do
        Begin
        Writeln('Do you want to Enter, Delete, Print, or Finish?');
        Readln(F);
        If F = 'Enter          ' then
                Enter(D);
        If F = 'Print          ' then
                Printdict(D);
        If F = 'Delete         ' then
                Delete(D);
        End; (* While *)
Writetodisk(D);
End. (* Program Stereo *)
```

Letters

Let me take a moment to encourage our readers to write in to us. We read every piece of mail and respond as fast as we can. I would also like to hear any comments and/or suggestions about the newsletter. What articles would you like to see? What programs would you like to try? I haven't heard a peep from you readers regarding these areas so I just wing it, hoping you'll enjoy the newsletter as is. We are very open to suggestions. I would also like to request you send in your programs, Pascal and Assembly Language. They are fun to read and I'm sure most of you readers have something to offer your fellow subscribers. After all Pascal isn't for wimps.....

On to the letters:

We have received many inquiries regarding our mysterious "Appendix H". I've decided to use Mel Holliday's letter as an example:

"Just received new update this a.m.. I'm going through the GREAT new manual I see a reference to Appendix H. I have no Appendix H. What happened to it? Of course that is the section I want to see."

Appendix H was originally a listing of the programs in our System Utilities Toolkit. It was removed at the last minute due to space limitations. Unfortunately, allof the references to it were not removed with it. For those of your who've already received the System Utilities Toolkit in the mail, I hope your enjoying it.

The next letter is from Neal Jensen of Beloit, Wisconsin.

"In the April issue of A+ there is a review of Pinpoint's Desktop-Accessory program. The article mentions that Pinpoint is working with Kyan and others to allow users to create accessories for Pinpoint using Pascal and BASIC. I am glad to see this cooperation between independent software houses. This cooperation should produce standards that will make a variety of software packages more compatible and easier to operate."

Funny you should mention this Neal, in our next issue we will discuss the use of Kyan Pascal with Pinpoint by Pinpoint Publishing and Macroworks by Beagle Brothers. I enjoyed your letter and want to thank you for writing in.

****

Rod Robichaux writes in about the "FirstWord" program in the tutorial (Tut. IV-30). As his letter was too long to include in the newsletter, I'd like to touch on the main point. Many people noticed the WHILE loop kept repeating and you couldn't exit. This was due to the way string handling is done in version 2.0. Once you've entered the word into a STRING ARRAY, the string gets packed with spaces after the word. You cannot compare a single character constant with the first letter of an ARRAY. You can, for example, compare a 15 character constant with a 15 character string. We felt this was very limiting to certain programs and have fixed this problem. Should any of you care to send us your disk we are alwayshappy to ship you a version which will access strings one character at a time. Thanks Rod for your nicely detailed letter!

****

The last letter I want to include in this column is from Kathleen Filer of Bowling Green, Ohio, who made some fine suggestions regarding our Pascal product. We DO pay attention to all suggestions made by our readers.

"There is a real inconsistency on pathname usage which could cause real confusion if a new user blindly follows your instructions. When does one need to change directories or volume names? When is it necessary to switch disks? Why is it necessary to use complete pathnames to name files in the editor? I suggest you clearly outline the two major modes of using Kyan Pascal - menu and KIX command.

"It took me a lot of time to figure out why sometimes the KIX commands worked alone, such as RM, and at other times the BIN/RM was required. I think you should clearly state that any file in the BIN directory on the booted disk (on which KIX.SYSTEM resides) or

in the present working directory, can be run with out typing the whole pathname. (Following your configuaration directions yields a second disk without a BIN directory, so any executable file would naturely run if its filename were entered.) Therefore, if a volume with KIX commands is booted, e.g. /KIX (D2,S1), the KIX commands are immediately available, even if the present working directory is changed to another disk. If the disk with MENU, INTRO, KIX, QUIT, and CD is booted, e.g. (D1,S1), any of those programs are immediately available by simply typing its filename, even if the present working directory has been changed to another volume.

"For one disk users this means that either the /Kyan.Pascal or the /KIX disk can be booted, depending on which mode the programmer plans to use - menu or KIX command. If the boot disk and the programming disk have the same volume name, no CD command is needed, and any command or file on the boot disk, including those in the BIN directory, OR any executable file on the second disk, on which there are no directories other than the volume name, can be accessed simply by typing its name. This even holds true if the second disk has a different volume name and CD has been used to change the present working directory to that second disk.

"For two-disk users this means that the /Kyan.Pascal disk is booted. However, its contents will depend on whether the user plans to use the menu mode or the KIX command mode. There is room for most of the KIX commands on that disk when MENU, INTRO, and FILER are removed. The boot disk should probably be renamed /KIX if it contains many KIX commands. Thus the original /KIX disk could be swapped in if an exotic command is needed. As the manual states, the first action after booting on the part of the user is to use the CD command to change the present working directory to /User, or whatever the volume name is for the storage disk.

"I certainly do not mean to discourage you with such a list of problems. I am glad to have Kyan Pascal and would encourage other Apple users to purchase it. (Even if the reviewer in inCider seemed to be only looking for a toy.) I hope these suggestions are useful "

Let me just add, Kathleen, that you need not type the entire pathname of the file when trying to use the editor. All you have to do is change your working directory with the CD command first, then type the filename.

UPDATE ... KYAN

Kyan Software Inc.
1850 Union Street #183
San Francisco, CA 94123

July/August Issue
Volume 1, Number 5

Apple Edition

(Editor: Sonja Newell)

WHAT'S NEW?

Attached to this issue of the newsletter is Kyan's catalog of products for the Apple //. Several new products are being introduced including:

- o TurtleGraphics Toolkit (Available now)
- o MouseGraphics Toolkit (Available 9/1/86)
- o Code Optimizer Toolkit (Available 8/1/86)
- o AppleWorks KIX (Available now)

The new version of KIX contains many improvements and enhancements, not the least of which is compatibility with AppleWorks. KIX is now a desktop accessory for AppleWorks. You can call the KIX window from within AppleWorks, execute any number of KIX commands or utilities, and return to AppleWorks right where you left off.

In the coming months, Kyan will introduce a series of utility programs which will run in the AppleWorks KIX environment. Also, a new programming toolkit will be available this Fall which will provide you with the utilities needed to write your own desktop programs for AppleWorks using Kyan Pascal.

Other improvements in KIX include:

- o A new command, MVV, now renames volumes.
- o CP and MV now have a -Q (Query) option, to prompt for permission to initiate copy processing before any takes place.
- o CAT and LPR now have -P and -F options: -P stops printing at the end of each page and asks for a new sheet; -F prints the filename being printed on the top of the first sheet printed.
- o CAT prints, displays and conCATentates AppleWorks Word Processor files. With the -V option active, you can view your AWP formatting options also.
- o CAT with redirection but no filenames lets you type into a TXT file or directly to the printer.
- o Compatibility with Catalyst, MouseDesk and other similar programs.
- o Pressing the [Escape] key aborts a command at any point in its execution.
- o Canceling output to screen, file, or printer by pressing [Escape].
- o ;*Type* fields now include AWP, ADB, and ASP file types.
- o You can specify file ;*types* in LS, RM, CP, and MV commands.

KIX UpGrade Options and Summer Software Sale!

There are two upgrade options for KIX. You can upgrade to AppleWorks KIX for only $20.00 (plus shipping and handling); this upgrade includes a new program disk, user manual and quick-reference card. Or, you can upgrade to KIX version 1.02 at no charge by returning your original KIX disk; this upgrade incorporates the non-AppleWorks changes listed on the previous page.

We would like to sell some Toolkits this summer and so we've put together an offer which you just can't refuse (we hope!). **If you order any Toolkit from our catalog before August 15, you can deduct 10 percent from the purchase price. Order two Toolkits and you'll save10 percent and get a free Kyan binder. If you order three or more Toolkits, you get a free binder plus a 20 percent discount. This offer expires August 15 so be sure to order right away! (Important Note: This special price offer is being extended only to subscribers to Update ... Kyan! If you phone-in your order, please be sure to mention that you are a subscriber and want the special prices.)**

Update

Programming Contest

After our last newsletter, we received numerous programs from you. I've come to the conclusion that there are some really great programmers using Kyan Pascal. I enjoyed ALL the submissions and encourage everyone to continue sending them.

The first winner of a $50.00 cash prize is Lester McCann. His winning entry is printed in the Assembly Language section of the newsletter.

Programming Notes

1. We have had a few inquiries regarding the mysterious compiler "error 47". The definition of this error is "Function not able to return structure in ISO". If you are trying to use a FUNCTION defined as an ARRAY, RECORD, pointer or anything other than simple ordinal types, you will probably run across this error. It is a restriction imposed by the ISO standard.

2. Another user asks that we list our latest versions in the newsletter so people will know when to send in their disks for an update. Something like this:

Latest versions (Apple):

Kyan Pascal Plus	2.02	
Kyan Pascal	2.02	
System Utilities Toolkit	1.00	(nothing new here)
MouseText Toolkit	1.00	"
Advanced Graphics Toolkit	1.00	"
TurtleGraphics Toolkit	1.00	"
KIX	1.02	(See "What's New")

We think this is a great idea because our subscribers don't have to rack up their phone bill calling to find out about our latest versions. So, beginning with this issue we will insert a box in the Update section with the latest revision level and the significant changes from the earlier version.

3. Mr. James Luther of Overland Park, Kansas has a suggestion for when a program drops you into the Apple monitor or you get an asterisk prompt after your program bombs. He suggests several calls which will save you from having to reboot the system. The following commands call the ProDOS MLI "QUIT" routine and take you back to KIX. There are different locations depending on which directives you are using.

Without using _SystemFile or _UsesHires directive, enter: 923G

With the _UsesHiRes, but not the _SystemFile directive, enter: 4123G

With the _SystemFile directive, but not the _UsesHiRes directive, enter: 8EEG

With the _UsesHiRes and the SystemFile directives, enter: 40EEG

Before calling these commands, you should check to verify that the address being called (923, 4123, 8EE, or 40EE) has the machine language instruction JSR $BF00 with a $65 after it by typing the address and a "L" (i.e. 923L)

If you cannot find any of these calls, enter:

```
800: 20 00 BF 65 06 08 04 00 00 00 00 00 00
800G
```

This program looks like this:

```
      JSR   $BF00     ;call MLI
      DB    $65       ;QUIT call
      DW    PQT       ;pointer to start of parameter table
PQT   DB    $04       ;4 parameters
      DB    $00       ;all 00's
      DW    $0000
      DB    $00
      DW    $0000
```

We like to print suggestions, technical notes, and answers to questions from Kyan users. If you have any, send them in and we will do our best to include them here in the newsletter.

Pascal Programming

Functions and Procedures in Pascal

by Ed Nelson III

The main difference between a function and a procedure is that a function must return a value, whereas a procedure does not. Procedures can, however, return a value. This article discusses the use of functions and procedures in Pascal programs.

A function will return the value in its name. For example, if a function is declared as "Function Paint: char;", then the word Paint will contain the character that has been returned by the function. This is limited in its usefulness; to retain the value returned by a function you must assign its value to a variable. However, functions are ideal for testing looping conditions since they provide the necessary information with a minimum of Pascal commands.

There are two types of procedures -- call by value and call by reference. The call by value procedure is declared like Demo3 in the demonstration program listed below. Although these procedures do not return values, they can affect global variables. Call by value procedures can be used to do anything that does not

require the return of information. For example, a call by value procedure can display information on the screen.

The other kind of procedure is the call by reference. The call by reference procedure is declared like Demo4 in the demonstration program. This kind of procedure returns its values in variables declared as VAR in the procedure heading. Once the procedure has completed execution, the variables contain their new values, where the information is retained until it is changed.

Note: ISO Pascal does not allow certain data structures to be returned by a Function (this is the 'Error 47' described elsewhere in this newsletter). A way around this is to use a procedure with variables passed by reference.

Functions and procedures have similar abilities. As a general rule of thumb, anything a function can do, a procedure can do, and vice versa. The largest difference between functions and procedures does not lie in their respective abilities and limitations, but in how they are used.

Functions are generally used to find and return a value. Procedures are used to do other things, such as displaying data, or opening and closing files. The main difference is stylistic, not something enforced by the language.

In the demonstration program below, functions and procedures are used for different aspects of pattern matching in strings. S2 will be the pattern that is looked for, and S1 will be the string that is searched. S3 is not used until it is passed into Demo4.

Note: There was a misprint in some early editions of the manual regarding the Index function. The correct description of this function is: the first string passed (S1) is the string to be searched. The second string passed (S2) is the pattern looked for in the first string.

The two functions, Demo1 and Demo2, are used to find a value and return it. Demo1 returns a TRUE if S2 is in S1 and returns a FALSE if S2 is not in S1. It is worth noting that a boolean function can be used wherever a boolean expression is used, such as in IF-THEN, UNTIL, and WHILE statements. For an example, look at the IF-THEN statement in the main program. The second function, Demo2, counts the number of occurrences of S2 in S1. The main program prints out the information received.

Procedure Demo3 is a call by value procedure. It does not pass any information back to the main program. Instead, it displays its information on the screen.

Procedure Demo4 is a call by reference procedure. This procedure breaks up S1 and return parts of it in S1, S2, and S3. After this procedure is called, S1 contains the part of the string before the pattern, S2 contains the pattern, and S3 contains everything after the pattern.

If you are going to pass large data types in a program, you may want to use a call by reference procedure rather than a function. The reason is that in a call by reference procedure, a point (that points to the data structure) is passed rather than the whole data structure (which is what happens with a function). Thus this technique saves time and stack space.

Following is the Demonstration program:

```
PROGRAM Demo (input, output);
    Const
        maxstring = 20;         {maximum string length is 20}
        empty = '               ';   {20 spaces}
    Type
        string = array [1..maxstring] of char;
    Var
            S1, S2, S3 : string;
            i : integer;
```

{The following function is an include file on the Kyan Pascal disk. String A1 is searched to see if it contains string A2. If it is, the location of the first character the string A2 is returned. If it isn't, a zero is returned. For example, if A1 is '*that theory is not viable at this time*' and A2 is '*viable*', the function INDEX (A1, A2) will return the value 20 which is the position of the first letter of A2 in A1.}

```
FUNCTION Index (VAR A1, A2: string): integer;
    VAR
        I, J, K, L: integer;
    BEGIN
        I := Maxstring;
        WHILE ((A2[I] = ' ') AND (I <> 1)) DO I := I - 1;
        K := 0;
     REPEAT
         J := 1;
         L := 1;
         WHILE (J <= I) DO
             BEGIN
                 IF (A1[J+K] <> A2[J]) THEN L := 0;
                 J := J+1;
             END;
         K := K+1;
     UNTIL ((L = 1) OR ((I+K) > Maxstring));
     IF (L = 1) THEN
        Index := K
     ELSE
        Index := 0;
    END;
```

{The next function is an include file on the Kyan Pascal disk. It returns the length of the string passed to it.}

```
FUNCTION Length (VAR A1: String): Integer;
    VAR
        I: Integer;
    BEGIN
        I := Maxstring;
        WHILE ((A1[I] = ' ') and (I <> 1)) DO I := I-1;
        LENGTH := I;
    END;
```

{The next procedure is also an include file on the Kyan Pascal disk. It copies part of A1 into A2, as specified by I (the position in A1 of the first character copied) and J (the number of characters copies to A2).}

```
PROCEDURE Substring (VAR A1, A2: String; I,J: Integer);
    VAR
        K : Integer;
    BEGIN
        FOR K := 1 TO Maxstring DO A2[K] := ' ';
        I := I - 1;
        FOR K := 1 TO J DO A2[K] := A1[I+K]
    END;
```

{This next function return a value TRUE if S2 is in S1, otherwise it returns a FALSE.}

```
FUNCTION Demo1 (S1, S2: String) : Boolean;
    VAR
        Result : integer;
    BEGIN
        Result := Index (S1, S2);
        IF Result = 0 THEN          {If result = 0, then S2 is not in S1}
            Demo1 := FALSE
        ELSE
            Demo1 := TRUE       {If result <> 0, then S2 is in S1}
    END;
```

{The next function returns an integer equal to the number of times S2 appears in S1. It will not work if the first character in S2 is a space.}

```
    FUNCTION Demo2 (S1, S2: string): Integer;
        VAR
            Result : integer;
            Dem : integer;
        BEGIN
            Result := Index (S1, S2);
                Dem := 0;
            WHILE Result <> 0 DO
                BEGIN
                        Dem := Dem +1;
                        S1 [Result] := ' ';
                        Result := Index (S1, S2);
                END;
            Demo2 := Dem;
        END;
```

{This procedure prints out the position and length of S2 in S1}

```
PROCEDURE Demo3 (S1, S2: String);
    VAR
        Result : integer;
    BEGIN
        Result := Index (S1, S2);
        Writeln (S1: length (S1));
        Writeln ('position: ', result : 2);
        Writeln ('length: ', length (S2) : 2);
    END;
```

{If S2 is in S1, this procedure will return everything before the first occurrence of S2 in S1 (in S1), everything after the first occurrence of S2 in S1 (in S3), and S2 itself (in S2). If S2 is not in S1, then S1, S2, and S3 are unchanged. For example, if S1 is '*Gumby and Pokey live*' and S2 is '*and*' (plus 17 spaces), then Procedure Demo4 will change S1, S2, and S3 into the following: S1 will be '*Gumby*', S2 will be '*and*', and S3 will be '*Pokey live*'. All three will be padded with spaces after the last letter.}

```
PROCEDURE Demo4 (Var S1, S2, S3: string);
    VAR
        Position, len, p, l : integer;
        temp : string;
    BEGIN
        Position := Index (S1, S2);
        IF Position <> 0 THEN
            BEGIN
                len := length (S2);
```

```
                    p := 1; l := Position -1;
                    Substring (S1, temp, p, l);
                    p := position + len; l := maxstring - p +1;
                    Substring (S1, S3, p, l);
                    Substring (S1, S2, position, len);
                    S1 := temp
                END;
        END;

BEGIN               {Main program]
    S1 := empty; S2 := empty; S3 := empty;
    S1 := 'Gumby and Pokey live';
    S2 := 'and                ';
    IF Demo1 (S1, S2) THEN          {Boolean function instead of Boolean expression}
        Writeln ('Demo1: It's there!');
    Writeln;
    Writeln ('Demo2: ', Demo2 (S1, S2));
    Writeln;
    Writeln ('Demo3: ');
    Demo3 (S1, S2);
    Writeln;
    Writeln ('Before Demo4:');
    Writeln ('S1: ', S1);
    Writeln ('S2: ', S2);
    Writeln ('S3: ', S3);
    Writeln;
    Demo4 (S1, S2, S3);
    Writeln ('After Demo4: ');
    Writeln ('S1: ', S1);
    Writeln ('S2: ', S2);
    Writeln ('S3: ', S3);
END.
```

Binary Search Tree Data Structure

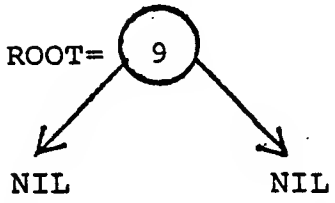
By Sonja Newell

A very fundamental Data Structure in Pascal is the Binary Search Tree. It is a widely used and talked about structure so perhaps many of you have heard of it before. This structure is useful when we have a set of records so large that it is impractical to use each element of the set as indices into arrays. An example would be a set of possible identifiers in a Pascal program.

Each node in the tree is a record or element in the set. The important property of a binary search tree is that all elements stored in the left subtree of any node are all less than the node itself and all elements stored in the right subtree are greater than the node itself. This makes for a fast search of your elements in a set.

A tree is made up of the "root" and it's "children". A node with no children is called a leaf. The "height" of the tree is the number of steps it takes to locate an element in a set. In order to keep the example simple I will use elements defined as integers when drawing a picture of the set.

The tree is first initialized to NIL. Then the first element becomes the root:

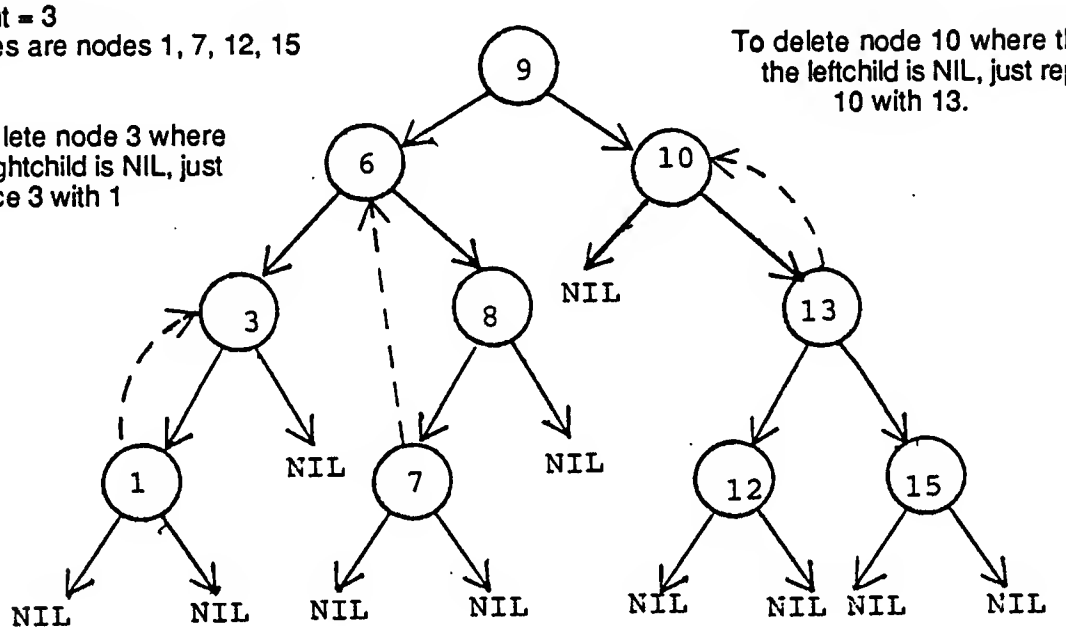


Children are inserted, with larger numbers to the right and smaller numbers to the left:

Height = 3
Leaves are nodes 1, 7, 12, 15

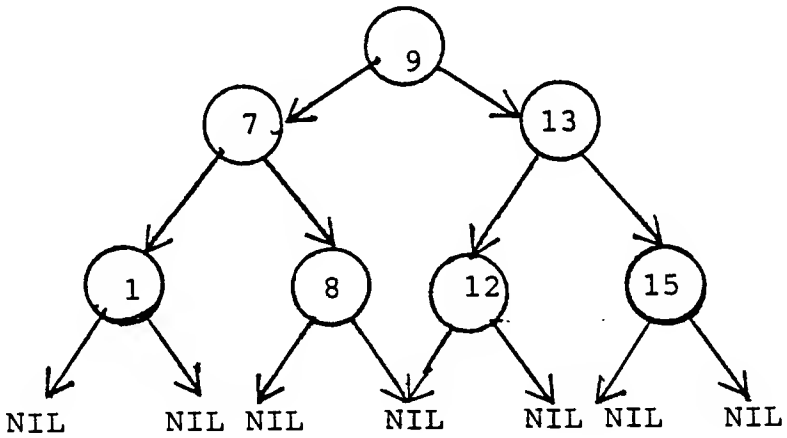
To delete node 3 where the rightchild is NIL, just replace 3 with 1

To delete node 10 where the the leftchild is NIL, just replace 10 with 13.



The leaves of the tree are easy to delete, but what happens when you try to delete a node with both children? This is why the DELETEMIN procedure is necessary. It works like this: Let's say we want to delete node 6. In the last statement of procedure DELETE we call DELETEMIN with a pointer to node 8 (the rightchild). The leftchild of 8 is not NIL, therefore DELETEMIN is recursively called with a pointer to node 7 (the leftchild). Node 7 has no leftchild and consequently node 6 is replaced by node 7 and the binary tree property still holds.

After deleting nodes 3, 10, and 6 we get a tree which looks like this:



I have written this program to demonstrate a binary search tree using a list of phone numbers. This is a very simplified example of trees. Notice the recursive procedures in this program. If you want to get complicated you can learn about other tree structures, such a ternary trees with more than two children, or how to balance a tree so all the nodes don't skew off to the left side and nothing on the right side of the tree.

I have once again used the book DataStructures and Algorithms by Aho, Hopcraft, and Ullman. Published by Addison-Wesley Publishing Company.

```
PROGRAM Phones (Input, Output);

(* This program demonstrates a binary search tree using a character array as the key field
   You can enter names and phone numbers into a record and put that record into the
   tree, find a record in the tree, or delete a record from the tree. *)

TYPE
        Name = ARRAY[1..20] of CHAR;
        Phnumber = ARRAY[1..8] of CHAR;
        PhSet = ^Nodetype
        Nodetype = RECORD
                        element: Name;
                        phone: Phnumber;
                        Leftchild, rightchild: PhSet
                End;

VAR
        PhoneSet: PhSet;
        Person: Name;
        Number: Phnumber;
        ch: CHAR;


FUNCTION Member (MemN: Name; MemA: PhSet): BOOLEAN;
BEGIN           (* Recursive function which checks for the element in a binary tree *)
        IF MemA = NIL THEN
        Member:= FALSE
        ELSE IF MemN = MemA^.element THEN
                Member:= TRUE
                ELSE IF MemN < MemA^.element THEN
                        Member:= Member(MemN, MemA^.leftchild)
                        ELSE IF MemN > MemA^.element THEN
                        Member:= Member(MemN, MemA^.rightchild);
END; (* function Member *)
```

```
PROCEDURE Insert (N: Name; Num: Phnumber; Var A: PhSet);
BEGIN           (* Add a record to the set *)
        IF Member(N, A) = FALSE THEN
        IF A = NIL THEN
        BEGIN
                NEW(A);
                A^.element:= N;
                A^.phone:= Num;
                A^.rightchild:= NIL;
                A^.leftchild:= NIL;
        END;
        ELSE IF N < A^.element THEN
                Insert(N, Num, A^.leftchild)
                ELSE IF N > A^.element THEN
                        Insert(N, Num, A^.rightchild);
END; (* procedure Insert *)


PROCEDURE Deletemin (VAR N: Name; VAR Num: Phnumber; VAR A: PhSet);
BEGIN           (* Deleting the smallest element *)
        IF A^.leftchild = NIL THEN
        BEGIN
        N:= A^.element;
        Num:= A^.phone;
        A:= A^.rightchild;  (* replace the node pointed to by node 'A' by its right child *)
        END;
        ELSE
        Deletemin(N, Num, A^.leftchild);
END;    (* procedure Deletemin *)


PROCEDURE Delete (N: Name; Num: Phnumber; VAR A: PhSet);
BEGIN           (* Removes node from the set *)
IF Member(N, A) = TRUE THEN
        IF A <> NIL THEN
                IF N < A^.element THEN
                        Delete(N, Num, A^.leftchild)
                ELSE IF N > A^.element THEN
                        Delete(N, Num, A^.rightchild)
                ELSE IF (A^.leftchild = NIL) and (A^.rightchild = NIL) THEN
                        A:= NIL
                ELSE IF A^.leftchild = NIL THEN
                        A:= A^.rightchild
                ELSE IF A^.rightchild = NIL THEN
                        A:= A^.leftchild
                ELSE BEGIN
                        Deletemin (N, Num, A^.rightchild);
                        A^.element:= N;
                        A^.phone:= Num;
                END;    (* else *)
END;    (* procedure Delete *)
```

```
PROCEDURE Find (N: Name; A: PhSet);
BEGIN
IF Member (N, A) = TRUE THEN
        Begin
        If N = A^.element then
                WriteIn ('Name: ',A^.element:20,'PH#: ',A^.phone:8)
        Else If N < A^.element then
                Find (N, A^.leftchild)
        Else If N > A^.element then
                Find (N, A^.rightchild);
        End (* If member *)
ELSE
        WriteIn ('Name is not in the set');
END;    (* procedure Find *)


BEGIN           (* Main Program *)
PhoneSet:= NIL; (* set the pointer to NIL *)
ch:= '#'; (* set flag *)
While ch <>'S' do
        Begin
        WriteIn ('Do you want to E)nter, F)ind, D)elete, or S)top?');
        ReadIn (ch);
        If ch <> 'S' then
                Case ch of
                        'E':Begin
                                WriteIn('Enter a name and a phone number');
                                ReadIn(Person);
                                ReadIn(Number);
                                Insert(Person, Number, PhoneSet);
                            End; (* 'E' *)
                        'D':Begin
                                WriteIn('Enter name to delete');
                                ReadIn(Person);
                                Delete(Person, Number, PhoneSet);
                            End; (* 'D' *)
                        'F':Begin
                                WriteIn('Enter name to Find');
                                ReadIn(Person);
                                Find(Person, PhoneSet);
                            End; (* 'F' *)
                End; (* case ch of *)
        End; (* While ch *)
END. (* program Phones *)
```

Assembly Language Programming

PROGRAMMING IN ASSEMBLY LANGUAGE WITH KYAN PASCAL 2.0 Article #3

by John R. Fachini

This installment of the assembly language column marks a change in the direction the column will be taking. Instead of trying to make half of each column for beginners and half for more advanced users, I'm going to select a topic of interest each issue and investigate exactly what's what with it. Usually both novices and experienced programmers will benefit. I don't mean to leave the beginners out in the cold;

this column, in a bi-monthly format, is too short to teach beginners anything they can't learn from a good 6502 book. I will, however, do my best to answer questions concerning the Kyan assembler and compiler if you write.

The topic I've chosen for this issue was suggested by Tom Osber in San Francisco, California. He wanted to know about the AS (6502 assembler) that is provided with the Kyan Pascal 2.0 package. This discussion will include working examples of assembler directives and macros also.

The Kyan AS (6502 assembler) provided with Kyan Pascal 2.0 is included as part of the package for two reasons.

1. To permit the programmer to include assembly code in Pascal sources, so that after the compiler has generated its macros, everything can be assembled into machine language.

2. To provide the user with a stand-alone, high power, easy-to-use assembler which adds to the programming capability with version 2.0.

In a future column we'll look at exactly how the compiler generates the macros which represent your Pascal program; for now we'll pursue using the assembler as a stand-alone facility.

At this point I'm going to assume a couple of things: first, that you are familiar with assembly code, and second, that you are familiar wit some of the terminology associated with assemblers. If you're not, get out your Kyan Pascal 2.0 manual and read chapter 5.

In any stand-along application, the assembler must first know the address of where the program will be loaded and executed. This is accomplished using the ORG statement:

```
ORG $800        ;Program will load and run at memory location $800.
```

(Remember that only labels can start in column 1.) The ORG has told the assembler to generate machine code that will execute starting at $800 in main memory. The remark to the right of the ORG statement is separated by a semi-colon (;). This is a good programming practice: always remark code, especially code that is special purpose.

I said a second ago only labels can go in column 1. True and false. Remarks can also go in column 1, as long as they start with a semi-colon. Some assemblers use an asterisk (*) in column 1 to represent a remark. We use the asterisk for something more important than that. Remember: only semi-colons are used to protect remarks.

The next directive of interest is the EQU directive. EQU assigns values to labels. Very often this makes for very easy reading of assembly language programs. For example, compare the next two lines:

```
JSR $FC58           ;Call Monitor ROM clear screen routine

JSR ClrScrn         ;Same as above
```

Obviously ClrScrn is a lot more obvious than "$FC58". There's another good reason for using labels in your program. If you decide to move your assembly code to another operating system, you only have to change one label definition and re-assemble your source code. If you reference memory locations by address (ex. $FC58), you'll have to change every reference to it before moving your code to another system.

Note: labels can be any length, but only the first 6 characters have "significance". For example, the labels "FlagItDone" and "FlagItUndone" are considered the same by AS since "FlagIt" appears in both. The assembler also ignores the cases of characters, i.e., "XYZ" is the same label as "xyz".

The next two directives are used when you need to generate tables of bytes or words in your assembly language code. Observe:

```
Data    DB      1,2,3,4,5
        DW      $4000,ClrScrn
```

The "Data Byte" (DB) and "Data Word" (DW) directives generate bytes in memory according to what follows. The above declarations would generate the following bytes at memory location "Data":

```
Data:  1  2  3  4  5  00  $40  $58  $FC
```

Notice that the "word" (2 byte) definitions are generated lo byte, hi byte. This convention is used on the 6502 micro processor and subsequently is maintained by the assembler whenever words are generated.

At times it is necessary to know the lo byte or the hi byte of a label value. This can be done using the >,<, and # directives:

```
        LDX #>ClrScrn       ;Load X with #$58 (lo byte)
        LDY #<ClrScm        ;Load Y with #$FC (hi byte)
```

These directives will prove very useful later on.

The "Data Storage" (DS) directive will reserve the number of bytes indicated in the object file. For example:

```
        DS  6
```

will skip 6 bytes in the object file being generated.

The next two directives allow you to generate character codes as tables in memory. They are "ASCii" (ASC) and "STRing" (STR). These directives perform the same operation with one exception: the string declared by STR starts with a length byte (the number of characters in the string); ASC defines one byte per character but does not generate a length byte. For example:

```
Info ASC 'HELLO'        ;Generate 5 bytes corresponding to the
                        ;ascii codes of the characters HELLO

More STR 'HELLO'        ;Generate 6 bytes:
                        ;       first byte = 5
                        ;remaining 5 bytes are the ascii codes of the
                        ;characters HELLO
```

Another important assembler character is the * (asterisk). The asterisk represents the 'current value of the assembler's program counter'. In other words, the * represents the hexidecimal value of the memory location in which the next instruction generated by the assembler will be put. The initial value of the * is set by the ORG directive.

The last directive we'll talk about in this issue is the MACRO directive. Macros are code segments which can be installed in the source code by referencing the macro name instead of typing the lines of code the macro is defined as. Another powerful feature of macros is their parameter passing ability. You can define a macro and use "parameters" to make the macro code being generated specific to the memory locations or labels you need to reference in "this" case of the macro in memory. For example:

```
PrtChar     MACRO           ;print a character
            LDA #'&1        ;Load A with first parameter in macro list
            ORA #$80
            JSR Cout
            ENDM            ;Mark END-of-Macro
```

The &1 represents the first parameter in the list used by the assembly code source. In the program, saying:

```
PrtChar A
```

would generate assembly code as follows:

```
LDA #'A
ORA #$80
JSR Cout
```

Remember that Macros are NOT subroutines! They only save time if the same code is used many times.

There are more directives to be discussed, but these are enough to digest for now. Next, we'll put all of these directives to use in a useful utility program.

Anyone who has ever had a bad experience with a floppy diskette knows how aggravating they can be. They tend to die at exactly the worst time possible. If you have a hard drive and are performing a backup of it, a bad diskette can ruin the entire procedure. So what I've got here is a program which will scan a disk and check for bad (unreadable/unwriteable) blocks.

Editor's Note: John's program is too long to print in this issue of the newsletter. If any of you would like a copy, just send a stamped self-addressed envelope to ***Update ... Kyan****. We will send you a copy of the 7 page listing.*

Programming Contest Winner

Mr. Lester McCann from Madison,Wisconsin sent us a great program which speeds up the DRAW and PLOT procedures of Kyan's HiRes.I file. The following describes his entry and lists the new routines to be included in a HiResALT.I file. We suggest you replace the appropriate routines in the HiRes.I file on your Kyan Pascal disks.

HGR -- This routine consists of a single line of machine code. All this routine does is call the Applesoft ROM routine located at $F3E2, which displays hi-res page 1 and clears it.

TX -- This routine is identical to the TX routine provided by Kyan in the file HiRes.I with a couple of comments added. It toggles two locations; the first sets text mode, and the second sets the text screen.

PLOT(X1, Y1, COLOR) -- this routine plots points on the hi-res screen. The format is the same as the HiRes.I PLOT routine, but the COLOR parameter is used differently. PLOT makes use of the Applesoft point-plotting subroutine to place a colored dot on the screen. Therefore, it uses the color numbering sequence familiar to users of Applesoft BASIC:

Color #	Color	Color#	Color
0	Black1	4	Black2
1	Green	5	Orange
2	Violet	6	Blue
3	White1	7	White2

To use PLOT, the X and Y coordinates of the point to be plotted are passed in, along with the desired color of the dot (in the range 0-7 given above). Notice that due to the different sequence of color values used by this version of PLOT, programs written to use the Kyan color sequence but recompiled to use HiResALT.I may have some color problems. If this is a serious problem, a short piece of code can be inserted at the front of the PLOT routine to convert Kyan color values to Applesoft color values. PLOT is rather heavily commented; anyone who has read and understood the section on assembly language programming in the Kyan manual should have no trouble following it. Location $F457 is the start of the plotting subroutine.

DRAW(X1, Y1, X2, Y2, COLOR) - This routine is used to draw lines on the hi-res screen. It makes use of the Applesoft ROM line drawing routine located at $F53A. As in PLOT, the color values are expected to be Applesoft values (PLOT actually takes care of setting the desired color). At first glance, it may seem silly to be calling PLOT to plot the first endpoint of the line. This is done to satisfy the built-in drawing routine, which assumes that the first endpoint of the line is already plotted on the screen. The drawing routine then takes the second endpoint and connects it to the first with a line of the given color.

When writing these routines, I tried to keep three things in mind - speed, size, and compatibility. I wanted these routines to be faster than their Kyan supplied counterparts, smaller than the Kyan routines(HiRes.I requires 10 blocks of disk space; HiResALT.I needs only 3), and compatible with the calling semantics of the Kyan routines. I gave priority to speed and size, which explains a couple of shortcomings. First, the color sequence used is the Applesoft sequence instead of the Kyan sequence. Depending on your point of view, this could be an advantage of these routines; if you're a former BASIC programmer, the old sequence is quite familiar. On the negative side, the same value gives one color in these routines and may give quite another in the Kyan routines. However, as mentioned previously, it can be changed with a small section of code. Second, there is no range checking done on the parameters of the PLOT and DRAW routines. This means that if a program calls these routines and passes an illegal value, it is possible that the program and perhaps the entire operating system will crash. Adding range checks is not a difficult task, but it would add quite a bit of code to the routines, and would slow them down considerably. As I was looking for speed and small size, I chose to leave out range checking.

```
PROCEDURE HGR;
Begin
#A
        JSR     $F3E2           ;display page 1 & clear it
#
End;

PROCEDURE TX;
Begin
#A
        LDA     $C054           ;set page 1
        LDA     $C051           ;set text screen
#
End;

PROCEDURE PLOT(X1, Y1, COLOR: INTEGER);
Begin
#A
        STX     _T
        LDY     #5
        LDA     (_SP),Y
        TAX                     ;X has color (0-7)
        JSR     $F6EC           ;set color
        LDY     #9
        LDA     (_SP),Y
        TAX                     ;X has LSB of X1
        INY
        LDA     (_SP),Y
        STA     _T+1            ;store MSB of X1
        LDY     #7
        LDA     (_SP),Y         ;accumulator has LSB of Y1
        LDY     _T+1            ;load Y with MSB of X1
        JSR     $F457           ;plot point (X1, Y1)
        LDX     _T
#
End;
```

```
PROCEDURE DRAW(X1,Y1,X2,Y2,COLOR: INTEGER);
Begin
        PLOT(X1, Y1, COLOR); (* plot starting point of line *)
#A
        STX     _T
        LDY     #7
        LDA     (_SP), Y
        STA     _T+1            ;store LSB of Y2
        LDY     #10
        LDA     (_SP),Y
        TAX                     ;X has MSB of X2
        DEY
        LDA     (_SP),Y         ;accumulator has LSB of X2
        LDY     _T+1            ;load Y with LSB of X2
        JSR     $F53A           ;plot to (X2, Y2)
        LDX     _T
#
End;

#A
_UsesHiRes
#
PROGRAM TRY(Input, Output);
Var
        I,J: Integer;
        ch: Char;
#i HIRESALT.I
Begin
HGR;
        for I:= 0 to 279 do
                for J:= 0 to 159 do
                        PLOT(i, j, 3)
        read(ch);
        TX;
End.
```

Speed comparisons:	HIRESALT.I	HIRES.I
	color = 3	color = 5
	52.0 sec	53.1 sec

```
#A
_UsesHiRes
#
PROGRAM TRY(Input, Output);
Var
        I,J: Integer;
        ch: Char;
Begin
HGR;
        for I:= 0 to 279 do
                DRAW(I, 0, I, 159, 3);
        read(ch);
        TX;
End.
```

Speed comparisons:	HIRESALT.I	HIRES.I
	color = 3	color = 5
	5.4 sec	21.5 sec

Letters

I am pleased to see such great response to my plea (and John's too) for you subscribers to write us in last months newsletter. I enjoyed ALL the comments (good or bad), suggestions, questions, and programs you sent in. So before I go any further, here is an example:

Thomas Buehner of Berlin, West Germany writes:

"When I have output words on the printer (with the help of PR.I) and I am switching back to an 80 column screen (also with PR.I), the screen is blank (as it does with DOS' PR#3 statement). In Nibble (April 1986. p115, MicroSPARC, Concord, MA) I found the following hint: "To return to 80 column mode without clearing the screen under ProDOS, type:

```
PRINT CHR$(4);  "PR#A$C307"."
```

When working with Kyan Pascal and KIX, a similar solution should be possible. Unfortunately, I don't know anything of ProDOS' internals. Could any of the readers help? And - a hint I read in the German Kyan "fanzine" (G. Foltin: Rest-Reboot-Bug. In: Kyan Rundschreiben #2 (15-May-86), Mr. George Foltin found out that KIX does not reboot any longer after hitting ctrl-reset, if you take the following provisions:

```
1.    Boot from disk 2, side 2 (BASIC.SYSTEM)
2.    BLOAD KIX.SYSTEM, A$2000, TSYS
3.    CALL  -151
4.    20B4:  20 6F FB    (here PWRUP would be destroyed)
5.    BSAVE KIX.SYSTEM, A$2000, L2183, TSYS...
```

Tom Donofrio from Ottawa, Ontario writes:

I have uncovered a potential problem with the assembler when a Pascal program whlch contains assembly functions or procedures, is compiled. If you use the same label in two or more of these functions or procedures, you will get a MULTIPLY DEFINED SYMBOL ERROR. This can be quite annoying because, for instance, if you use the INCLUDE file HiRes.I in your program, you will not be able to use the labels I or J in any of your assembly language routines. Thus, two routines which behave perfectly well by themselves might run into problems when combined together in the same program. It seems to me that provisions for local labels should be provided. In the meantime, I have resorted to using as few labels as I can get away with.

I would like to make a suggestion to other users of Kyan Pascal Plus. As you know., the program disks come with Apple's ProDOS FILER program. This can be used instead of the KIX system for basic file manipulation. My suggestion is to dump it and go out and buy Central Point Software's Copy II+, version 6.0 or higher. These guys were producing intuitive, easy to use software long before the Macintosh came along. The program called UTIL.SYSTEM on the Copy II+ disk is the best ProDOS disk utilities I know of. It does all that FILER does and much more, plus it only occupies 53 disk blocks. I have placed this program on my KYAN.PASCAL disk along with as many KIX commands as I could fit and found that it works well In the KIX environment. The only problem I've found is that you have to warmstart the system in order to get back into KIX from UTIL.SYSTEM. A fix for this would be welcome.

Editor's Note: This will be fixed in version 1.1 of KIX due out this month.

Cheryl Thompson of Omaha, Nebraska writes to us about her RAMWORKS II Expansion card:

The following boot disk was designed after reading information accompanying the RAMWORKS II Expansion card, and the Kyan Pascal manual. Start with a ProDOS initialized disk, volume name Kyan.Pascal. Next copy (in order) ProDOS, BASIC.SYSTEM, and PRODRIVE for the desktop Expansion disk accompanying RAMWORKS II. Next copy KIX.SYSTEM, P.OUT, and A.OUT from the Pascal system disks to the boot disk you are creating. You will need to rename these files with a different volume name and use an intermediate disk as you can't copy from one disk to another disk with the same volume name. Next create a subdirectory named BIN. Then copy the following files from the Pascal system disks using the pathname of KyanPascal/BIN: STDLIB.S, AS, ED, PC, FILER, and CD. Finally create and save the file called "startup". This program is listed in the RAMWORKS II book. The only change required is that you must insert KIX.SYSTEM into the program instead of APPLEWORKS.

This disk can then be used as your boot disk. It will automatically boot Pascal and load the extra memory. It should not be used by beginning Kyan Pascal users as the disk does not have room for the HELP file or files such as QUIT. It does contain most of the programs that you will need during program production. The PRINT program will have to be obtained from the USER disk. I have been using this system for a month now and have had no problems. If someone does have a problem or has a better idea please let me know. Thank you for your consideration, and your help on the phone that got me started towards solving this problem.

Well, Cheryl, as you know, we try our best to answer phone calls right way or get back to our customers if we don't know of a solution immediately. I think you have some very useful information for our readers here a nd, if anybody would like to comment on Cheryl's boot disk set-up, we will be more than happy to print it in the next newsletter. It is nice to hear we could help you out. Thanks to everyone for writing.

Editors Note

We would like to thank all of you who returned the reader survey card included in the last issue. If you haven't, there's still time to make your opinions known. Please fill out the card and return it today. In the next issue we will publish a summary of what people had to say.

UPDATE ... KYAN

Kyan Software Inc.
1850 Union Street #183
San Francisco, CA 94123

September/October Issue
Volume 1, Number 6

Apple Edition

WHAT'S NEW?

Finally -- the New, Improved Apple II!!!!!

The long-awaited 16-bit Apple IIX will be formally announced by Apple Computer this month. Due to our non-disclosure agreement with Apple, we have been unable to tell you much about it (and still can't until they formally announce it on September 18). However, we have had two of the prototypes since the first of the year, and I can guarantee that you are going to love the new machine. The next issue of the newsletter will contain extensive coverage of the new machine and the products which Kyan will be introducing to support it. In the meantime, let us wet your appetites a little.

The Apple IIX is based on the 65816 microprocessor running at 3 MHz (3 times faster than the current Apple II). It will operate in three modes -- straight 6502 emulation; fast 6502 emulation (3 MHz); and 16-bit mode. Almost all existing Apple II software will run in one or both of the 6502 modes, and you can expect developers to roll out updated versions of existing packages and some totally new products which will run in 16 bit mode. The standard IIX includes a detached keyboard (like the Mac PLUS), high resolution color graphics, a dedicated sound chip, 128K of ROM (you MouseText and MouseGraphics programmers will find the runtime modules in ROM!), and a capacity for more than 4 meg of RAM (units will be shipped with 256K). Apple will also announce an upgrade program which allows Apple II owners to upgrade their systems by replacing the motherboard. The October issue of InCider will contain a full description of the Apple IIX.

Kyan has been using the prototypes to develop a new version of KIX, a new macro assembler, a modified Pascal compiler, and modified programming toolkits for the machine. We will provide full specifications and exact availability dates in the next issue of the newsletter. All of your Kyan Pascal code written to date will port directly over to the new machine. If you think your Pascal routines are fast now, just recompile them with our new 16-bit compiler and run them at 3 MHz! Then try to figure out how you are going to slow them down again!

Toolkit VI Code Optimizer

The Code Optimizer Toolkit is designed for the advanced programmer who needs to reduce the code size of an application program and/or increase its runtime speed. The Toolkit consists of two modules -- the Code Optimizer and Source Code for the Kyan Runtime Library. The Optimizer performs two major functions:

1. It modifies the intermediate macro file generated by the compiler ("P.OUT") so that the Assembler generates "Program Specific Code" (i.e., code which includes only those Runtime Library routines which are specifically required by the program).

2. It replaces certain combinations of compiler-generated macros with optimizer macros which shorten code size and increase the runtime speed of the application program. (The areas most improved by the Optimizer are global variable accesses and record field calculations.)

The Optimizer can reduce the size of a program by more than fifty percent and, in some cases, almost the double execution speed. The following results were achieved using the Code Optimizer on the Sieve of Erotosthenes.

Program	Compiled Only	Optimized	Improvement
Code Size	12.9K Bytes	3.1K Bytes	9.8K
Runtime Speed	15 seconds	5 seconds	10 seconds

Source code for the Runtime Library also offers the programmer the following advantages:

1. The separate Kyan Pascal Runtime Library is not longer required on the disk. Now the application and the Pascal Runtime Library routines are combined in a single executable file.

2. The programmer can customize library routines (written in assembly langauge source code) to optimize performance and/or meet the specific needs of an application program.

The Code Optimizer Toolkit is a valuable tool for those programmers who are writing large applications and/or who are using MouseText, MouseGraphics, or Advanced Graphics routines in their programs. The Toolkit requires Version 2.0 of Kyan Pascal and can be used on any Apple II with 64K It retails for $149.95 (plus shipping).

COMPUTER GRAPHICS: A Programming Approach

This 445 page textbook written by Steven Harrington and published by McGraw-Hill is the perfect accompaniment to Kyan's Advanced Graphics Toolkit. It provides the hands-on experience and basic information needed to implement, modify, and use a computer graphics system. The book is built around detailed language independent algorithms for a graphics system and follows the standards proposed in the Graphics Standards Planning Committee's CORE system. By using this standard of basic graphics capabilities, the book provides a solid foundation for more advanced techniques. It includes coverage of raster graphics and discusses interactive techniques, enabling the reader to learn methods of graphical input as well as output. In addition, general 3D viewing is treated to familiarize the reader with the CORE system approach to viewing three-dimensional objects. Numerous problems and experiential exercises are included to enhance comprehension of the material.

For those users of the Advanced Graphics Toolkit who don't want to miss the additional support provided by this great textbook, you can order it by calling Kyan or mailing your payment. The price is $36.95 (plus 4.50 for shipping and handling).

UPDATE

Programming Contest

I believe we were right in our conclusion that there are some great programmers using Kyan Pascal. We have again received many valuable programs from you. Just as a reminder for the new subscribers of Update ... Kyan, we will pay each winner of our programming contest $50.00 in cash. But, our staff wants to ask you to do them a big favor. It would be great if you would submit your programs on disk rather than as a listing. Since we receive so many listings from you, we have problems entering all of them in order to run them. Thus it would be greatly appreciated if you would copy your programs on a disk.

This month's winner of our programming contest is Tom Donofrio. His winning entry is printed in the Assembly Language section of this newsletter.

Software Revision Status

Product	Revisions	
Kyan Pascal	2.02 (A)	(See below)
KIX	1.02	(no change)
System Utilities Toolkit	1.00	(no change)
MouseText Toolkit	1.00	(no change)
AdvancedGraphics	1.00	(no change)
TurtleGraphics	1.00	(no change)
MouseGraphics		(available 10/15)
Code Optimizer	1.00	(just released)

Changes to Kyan Pascal in 2.02A:

a) SEEK now handles up to 16 million bytes in a file.
b) Integer multiplication now correctly handles overflows.
c) Addition greater than 32767 now results in an arithmetic overflow.
d) Handling of local and global variables has been improved; functions can now be called that generate values which can be directly written to a text file.
e) Recursive creation of files now works.
f) When using global files, a procedure which creates a local file now leaves the global reference numbers intact (previously, they were destroyed).

IMPORTANT NOTE REGARDING PRODUCT UPDATES: Due to the volume of product update requests and the increasing costs associated with handling them, Kyan must implement a new policy regarding updates. Effective September 1, 1986, the following policy is in effect:

> **Within 90 days of purchase, customers may receive a product update at no charge by returning the original Kyan source disk to Kyan. After 90 days, customers may update their software by returning the original Kyan disk and a $10.00 handling charge.**

We regret this change in policy but hope you understand the need for doing so.

Reader Survey Results

The response to the reader survey in the June/July issue of Update... Kyan has been great. The survey has proved to be a valuable source of information for us. In the future, we definitely will consider the indicated areas of interest in our product development as well as in the design of the newsletter.

We thought that you would be interested in learning what other readers of the newsletter are interested in and so we are publishing the results.

New Product Interests

Product	Very Interested	Somewhat Interested	Not Interested	No Opinion	Total Respondents
Languages					
Modula-2	43%	29%	12%	17%	248
C	51	34	7	8	257
ProLog	17	30	25	28	246
LISP	20	37	27	17	236
FORTRAN	23	25	45	7	241
BASIC	9	17	66	9	234
ADA	23	31	27	19	239
Toolkits					
Animation	38%	38%	13%	11%	225
More Utilities	55	37	2	7	241
Data Base	43	32	15	10	239
Text Editor	36	33	23	9	236
AppleWorks Macros	20	19	39	22	228
Pinpoint Macros	14	22	31	34	222
Telecom.	39	31	18	2	234

Newsletter Interests

Type of Article	How many should be published in the newsletter? Many	Some	None
General Interest	20%	69%	11%
Technical	77	22	2
Pascal-oriented	82	18	0
Assembler-oriented	52	45	3
Programming Tips	57	38	5
Program Listings	43	56	1
Tech. Notes/Bugs	78	21	1
Industry Trends/Gossip	21	61	18
How people use K.P.	20	67	13
Software Development, Licensing, marketing	20	63	17

Do you own a modem?

YES	169
NO	97

If yes, are you interested in using EMail to communicate with Kyan?

YES	108
NO	57

If no, are you planning to purchase a modem in the next 6 months?

YES	40
NO	57

If no, why don't you want to purchase a modem?

Too expensive	25
Too difficult to use	2
Just don't see value	30

We have selected 10 winners from the list of those returning the survey questionnaire. Each of the following have won a free programming toolkit.

Ralf Augspurger, Neckarsteinach, West Germany
Matthew J. Chichester, Aurora, CO.
Rolf Kewitz, Bonn, West Germany
John Mills, Zachary, LA.
David E. Newman, New York, N.Y.
Matthew Palmer, Raleigh, N.C.
Michael W. Rutherford, Houston, TX.
Jeff Schumaker, Gibsonburg, OH.
John Wengert, Syracuse, N.Y.
Craig Winters, Merrit Island, FL.

Congratulations, you will receive a letter from us asking which toolkit you would like. And thanks again to all of you who mailed in your reader survey card.

PASCAL PROGRAMMING

Using a Kyan Pascal (Version 2.0) Program as a .SYSTEM file
by John Fachini

Many programmers have used Kyan Pascal to write 'startup' or ProDOS .SYSTEM files. A .SYSTEM file is a special type of file because it is the first file executed after PRODOS is loaded and in place.
Kyan Pascal is a great language with which to develop sophisticated .SYSTEM files. However, the process which must be followed to get the volume booted and the KYAN .SYSTEM file running is not clearly as documented in the manual; it is the purpose of this article to clarify and provide a useful program as an example of this process.

A Little Background
When you boot a diskette, the hardware in the Apple loads block #0 of the diskette and attempts to execute it. When the disk has been formatted (using KIX's FORMAT or the ProDOS FILER), this block of code searches the volume directory for a file named ProDOS. If it cannot find it, the code displays the message:

*** UNABLE TO LOAD PRODOS ***

Assuming everything is OK to this point, the PRODOS file is loaded and relocated in segments into the language card area of main memory, leaving the rest of main memory free for code and data.

Now that the PRODOS file has control of the system, it searches the volume directory it came from for a file with a name ending in .SYSTEM and which has a file type of $FF (225 decimal). Once this file is found, it is loaded starting at location $2000 in main memory and then executed.

The great part about using Kyan Pascal to write your .SYSTEM file is that you have the power of Pascal, assembly language, and all of the Kyan Toolkits at your fingertips; once you get your stand-alone disk booted, the world is all yours.

The Sample Program

The program I have included with this article is a good example of how to use Kyan Pascal to generate a .SYSTEM file. However, as you are typing the listing in, you'll probably have to change the procedure named "INIT_ARRAY". Here's what you should do for INIT_ARRAY:

> Every .SYSTEM file you want to display as an option on the startup screen should have it's filename assigned to an element in the StartupFiles array. Be sure that that all of the unused entries are initialized to 'Blanks', and the integer variable MAX is the number of entries to display from the array data you have provided.

The program is very straight forward. The important part is the procedure to follow when implementing this .SYSTEM file on a stand-alone disk.

1. Boot into KIX as usual.
2. Change the directory (CD) to one to work in (we'll call it WORK):
   % CD WORK
3. Type in the program and save it as file "BOOTUP.P"
4. Compile the program:
   % PC BOOTUP.P
5. Format a floppy disk. For now, name it "/TESTVOL":
   % FORMAT (6,1) /TESTVOL
6. Copy the BOOTUP program onto the new volume, renaming it with a .SYSTEM suffix at the same time:
   % CP BOOTUP /TESTVOL/BOOTUP.SYSTEM
7. Go back to the home volume:
   % CD
8. Copy PRODOS and the Pascal LIB also onto the disk:
   % CP PRODOS BIN/LIB /TESTVOL

That's it for testing purposes. Now boot your /TESTVOL and see for yourself. Once you're convinced, you can either use the demo program or use the precedure for your own startup program.

Enhancements

I am sure that the program I have written won't replace Catalyst or MouseDesk. It could use a few improvements I have left out. For example:

- Use the System Utilities Toolkit's GETDIR routine to scan the boot volume's directory for all .SYSTEM files (this way you can add more without modifying the source code).

- Set the program up to run in 80 columns.

- Search subdirectories for .SYSTEM files also.

I am glad to see people are really stretching Kyan Pascal to help them with all their system's tasks (not just programming, but system maintenance, control, and organization). Between KIX and Pascal, you have a lot of resources to call on. I love to see what you clever folks are up to.

Anyway, here's the program. See you at the Assembly Language column:

```
#a
_SystemFile
#
PROGRAM Demo_System_File;

CONST
  Blanks = '               ';  (* 15 blanks *)
  Left Arrow = 8;
  RightArrow = 21;
  DownArrow = 11;
  UpArrow = 10;
  ReturnKey = 13;
  EscapeKey = 27;
  MaxFiles = 10;

TYPE
  Name_String = ARRAY [1..15] OF CHAR;
  Name_Array = ARRAY [1..MaxFiles] OF Name_String;

VAR
  StartupFiles : Name_Array;
  Index,Max : INTEGER;
  Quit : BOOLEAN;

PROCEDURE HOME;
BEGIN
#a
  stx _t
  jsr $fc58
  ldx _t
#
END;

PROCEDURE GOTOXY (x,y:INTEGER);
(* This is a quick and dirty 40-column gotoxy routine *)
BEGIN
#a
  stx _t
  ldy #7
  lda (_sp),y
  sta $24
  ldy #5
  lda (_sp),y
  sta $25
  jsr $fc22
ldx _t
#
END;

PROCEDURE INVERSE;
BEGIN
#a
  lda #63
  sta $32
#
END;

PROCEDURE NORMAL;
BEGIN
#a
  lda #$ff
  sta $32
#
END;

FUNCTION KEYPRESS:INTEGER;
VAR key:INTEGER;
BEGIN
#a
  ldy #5
Kp lda $c000
  bpl Kp
  bit $c010
  and #$7f
  sta (_sp),y
  iny
  lda #0
  sta (_sp),y
#
  keypress:=key
END;

PROCEDURE INIT_ARRAY;
(* Put your system file name in here! *)
BEGIN
  StartupFiles[1] :='KIX.SYSTEM                ';
  StartupFiles{2] :='APLWORKS.SYSTEM' ;
  StartupFiles[3] :='BASIC.SYSTEM ';
  StartupFiles[4] :='BACKUP.SYSTEM';
  StartupFiles[5] :=Blanks;
  StartupFiles[6] :=Blanks;
  StartupFiles[7] :=Blanks;
  StartupFiles[8] :=Blanks;
  StartupFiles[9] :=Blanks;
  StartupFiles[10]:=Blanks;
  Max:=4;          ( # of files in list)
END;

PROCEDURE SHOW_MENU;
VAR loop:INTEGER;
BEGIN
  HOME:
  WRITELN ('Kyan Pascal v2.0 Startup demonstration'); WRITELN;
  WRITE  ('Use the arrow keys to scroll through the');
  WRITELIN ('list; press RETURN to select or ESC to');
  WRITELN ('exit to ProDOS...');WRITELN;
  WRITELN ('    System File    ');
  WRITELN ('------------------------------ ');
  FOR loop:=1 to 10 DO WRITELN('  ',StartupFiles[loop])
END;

PROCEDURE HIGHLIGHT (index:INTEGER);
BEGIN
  GOTOXY (3,index+7);
  INVERSE;
  WRITELN (StartupFiles [index]);
```

```
  NORMAL
END;

PROCEDURE CLEAR (index:INTEGER);
BEGIN
  GOTOXY (3,index+7);
  WRITELN (StartupFiles[index])
END;

FUNCTION PROCESS_MENU  :INTEGER;
VAR index, oldindex, ch  :INTEGER;
      done               :BOOLEAN;
BEGIN
  index:=1;
  REPEAT
        HIGHLIGHT(index);
        ch :=KEYPRESS;
        IF ((ch=LeftArrow) OR (ch=DownArrow)) THEN
        BEGIN
              oldindex:=index;
              index:=index-1;
              if index = 0 then index:=Max;
              CLEAR(oldindex)
        END;
        Done:=(ch=ReturnKey);
        Quit  :=(ch=EscapeKey);
  UNTIL (Done or Quit);
  PROCESS_MENU:=INDES
END;

PROCEDURE EXECUTE (VAR PathName:Name_String);
VAR error:BOOLEAN;
BEGIN
#a
   stx _t
   jsr _mli
   db $c7                ;getprefix
   dw GetPfx
   bne x9
   ldy #1
x1 equ *
   iny
   lda $281,y
   cmp #'/
   bne x1
   iny
   sty $280              ;length of boot volume's name
   ldy #6
   lda (_sp),y
   sta _t+1
   iny
   lda (_sp),y
   sta _t+2              ;address of pathname
   ldy #0
   ldx $280
x2 equ *
lda (_t+1),y
   cmp #32               ;blank marks end of word in array
   beq x3
   inx
   iny
   sta $280,x
   cpy #15               ;end of array entry?
   bne x2
x3 equ *
   stx $280              ;new length
   jsr _mli
   db $c4                ;get file info
   dw Parmlist
   bne x9                ;error
   lda ParmList+4
   cmp #$ff              ;system file?
   bne x9
   ldy #0
x4 equ *
   lda RelCode,y
   sta $be00,y           ;relocate loader call
   iny
   bne x4                ;move one page (more than enough)
   jsr $be00             ;use JSR in case of an error
x9 equ *                 ;error
   lda #1
   ldy #5
   sta (_sp),y           ;flag true boolean
   ldx _t
#
   IF error THEN
   BEGIN
        GOTOXY(0,20);
        WRITE(Cannot load.SYSTEM file.  Hit RETURN');
        READLN
   END
END;
#a
ParmList db 10
   dw $280
   ds 15
GetPfx db1
   dw $280
;
RelCode equ *            ;this code gets moved to be$00
   jsr _mli
   db $c8                ;open
   dw OpenFile
   beq *+3
   rts                   ; return if error
   lda OpenFile+5        ;reference number
   sta ReadFile+1
   sta CloseFile+1
   jsr _mli
   db $ca                ;read
   dw ReadFile
   pha                   ;save result
   jsr _mli
   db $cc                ;close
   dw CloseFile
```

```
  pla
  beq *+3                ;no error
  rts
  ldx #$ff
  txs                    ;reset the stackpointer
  jmp $2000              ;and execute theSYS file
;
OpenFile equ *-RelCode+$be00
  db 3
  dw $280
  dw$b000                ;open file buffer address
  db 0
;ReadFile equ *-RelCode+$be00
  db 4,0
  dw $2000,$ffff,0
;
CloseFile equ *-RelCode+$be00
  db 1,0
#

BEGIN
        REPEAT
                INIT_ARRAY;
                SHOW_MENU;
                index:=PROCESS_MENU;
                IF NOT quit THEN EXECUTE(StartupFiles[index]);
        UNTIL quit
END.
```

ASSEMBLY LANGUAGE PROGRAMMING

PROGRAMMING IN ASSEMBLY LANGUAGE WITH KYAN 2.0
Article #4
by John R. Fachini

This article will cover an issue may programmers have raised with Kyan Pascal in the past--that of runtime errors and being unable to trap them. Like any other language, Pascal has a great number of potential fatal errors. In most cases, like numeric overflows or stack overflows, the error MUST be fatal since the environment in which the program is running has been corrupted by the error.

Other errors, however, are non fatal. Most of these errors are related to disk access. Trying to RESET a file when it doesn't exist is not always fatal; often, it would be useful to trap this error as it occurs. But, since this isn't BASIC, there is no ONERR GOTO construct. That's why we have assembly language built into Kyan Pascal, and why we have spent a lot of time developing toolkits for you folks. The System Utilities Toolkit provides more than two dozen ProDOS functions and procedures. OK -- I am not advertising, just making a point. There is a solution. But...

The program which I have included with this column contains a function named IORESULT. In order to keep life simple, it returns the 5 most common ProDOS error messages as integers between 1 and 5. Zero denotes no error. IORESULT does not tell you what type the file is, if it is open or not, if is is damaged or not, or anything else. IORESULT answers the question - "Is the file in the correct directory on the correct volume accessible at this time?"

The demo program uses an array which will verify the IORESULT values. Change the array values in the INIT_ARRAY procedure and give the test program a try.

I hope this solves some problems. I realize it isn't a catch-all but one thing at a time.

If you haven't read the article in this newsletter about .SYSTEM files go read it and come back. Good. Now, I made references in that article about hearing from you people and what you are doing. I have received a few letters and I appreciate the good words. I don't mind hearing complaints or what you might think of as 'stupid questions'. No question is stupid. Those of you who have talked on the phone with me know (at least I hope) that I take everyone seriously and I do my best to help solve your problems. I can't debug programs for you, but if it relates to Apple hardware or the Pascal, KIX, or Assembler environment, I am happy to help. I am almost tempted to give you my CompuServ number, but I am afraid too many of you will recognize me from the CB as the guy with the lamp shade on his head. (Just kidding).

Here's the program. See you in November!

```
PROGRAM IORESULT_DEMO;
CONST
     MAX=6;

TYPE
     Pathstring = ARRAY [1..65] OF CHAR;
     Patharray  = ARRAY [1..MAX] OF Pathstring;

VAR
     PathName : Pathstring;
     Paths    : PathArray;

FUNCTION IoResult (VAR Pathname:Pathstring): INTEGER;
(* This functions returns a simplified error code for checking the existance of a file.
 Values returned are:

      0  no error
      1  file not found
      2  subdirectory not found
      3  volume not found
      4  invalid pathname
      5  volume appears damaged
 any other value returned is the
ProDOS MLI Error code *)

VAR Result :INTEGER;
BEGIN
#A
  stx _t
  ldy #9          ; address of the PathString
  lad (_sp),y
  sta _t+1
  iny
  lda (_sp),y
  sta _t+2
  jsr ConvPath    ; make into a ProDOS Pathname
  lda _t+1        ; put the pathname's address in the parameter list
  sta PList+1
  lda _t+2
  sta PList+2
  jsr _mli
  db $c4          ; Get_File_Info
  dw PList
  beq IOE9        ; no problem...
  ldx #0          ; table index
IOE2 equ *
  cmp IErr,x
  beq IOE3
  inx
  cpx #5          ;no more to check?
  bne IOE2
  beq IOE9        ; ACC=error code already
IOE3 equ *
  inx
  txa             ; A=new error code for this function
IOE9 equ *
  ldy #5
  sta (_sp),y     ; give function value to RESULT
  lda #0
  iny
  sta (_sp),y
  jsr FixPath     ; put the PathString back the way it was
  ldx _
#
  IoResult:=Result
END;
#a
PList db 10
  ds 17
IErr  db $46,$44, $45, $40, $5a
;
ConvPath equ *
  ldy #64         ; count chars backwards
CP1 equ *
  lda (_t+1),y
  cmp #32         ; blank?
  bne CP2         ; no: found last char in the pathname
  dey
  bpl CP1
  iny             ; make y=0
  tya
  sta (_t+1),y    ; makes length of pathname zero
```

```
CP2 equ *
   iny              ; actual path length
   sty _t+4
   dey
CP3 equ *
   lda (_t+1),y
   iny
   sta (_t+1),y     ; move string data forward by one byte
   dey
   dey
   bpl CP3
   iny              ; y=)...this is where the length byte goes
   lda _t+4
   sta (_t+1),y
   rts              ; done for now
;
FixPath equ *
   ldy #0
   lda (_t+1),y     ; path length byte
   sta _t+4         ; save it
   iny
FP1 equ *
   lda (_t+1),y
   dey
   sta (_t+1),y     ; move the string back this time
   iny
   iny
   dec _t+4
   bpl FP1
   rts
#

PROCEDURE HOME;
BEGIN
#a
   stx _t
   jsr $fc58
   ldx _t
END;
```

```
PROCEDURE TEST_RESULTS;
(* This routine calls IO.RESULTS for each pathname
 in the Paths and prints the resulting error code. Use this
 to test for yourself the IORESULT routine *)

VAR loop:INTEGER;
BEGIN
     WRITELN('*** IOResult Error Demo ***'); WRITELN;
     WRITELN('Function values returned: ');
     WRITELN('------------------------------')
     WRITELN('          2        Directory not found');
     WRITELN('          3        Volume not found');
     WRITELN('          4        Invalid pathname');
     WRITELN('          5        Volume appears damaged');
     WRITELN('          other    ProDOS Error code'); WRITELN;
     WRITELN('Error     /        Pathname');
     FOR loop:1 TO MAX DO
        WRITELN(IORESULT(Paths[loop]):3,'        '.Paths[loop]);
END;

PROCEDURE INIT_PATHS;
(* Initial pathnames in the Paths array *)
BEGIN
  Paths[1]:='/hard1/anyfile
  Paths[2]:='/hard1/bin./anyfile
  Paths[3]:='/vol.not.here
  Paths[4]:='this.pathname.is.too.long
  Paths[5]:='&.invalid.name
  Paths[6]:='/hard1/bin
END;

BEGIN
     HOME;
     INIT_PATHS;
     TEST_RESULTS
END.
```

Shape Routines
by Tom Donofrio

Tom Donofrio from Ottawa, Ontario has sent us a group of routines (to be put in an include file named **Shapes.i**) that allow you to draw shape tables using Kyan Pascal. Programs using it require the files **HiRes.i** and **BLoad.i** to be included (**BLoad** is from the System Utilities Toolkit).

The first routine to be called should be **SetShapes**, which sets aside some room in high memory and initializes the shape table pointer. It then loads the specified shape table into memory (via **BLoad**). You can use any of the hundreds of vector shape tables that are presently available for the Apple, as long as they are smaller than 4096 bytes (you can always check the size of a table by looking at its extended directory).

The next procedure is **SetRSC**, which sets the rotation, size, and color of the shape The following two procedures, **DrawShape** and **XDrawShape**, are analogous to the Applesoft DRAW and XDRAW commands.

We are also including a demo program from Mr. Donofrio which tests the shape drawing routines. It loads the shapes from a file named **ASCII.Shapes**.

```
(*            SHAPES.I              *)
(* ROUTINES FOR USING SHAPE TABLE. *)

FUNCTION  SETSHAPES(SHAPETABLE:PATHSTRING):INTEGER;

(* THIS FUNCTION SETS UP THE POINTER TO THE SHAPE TABLE AND LOADS IT
   INTO MEMORY.  IT RETURNS A MLI ERROR CODE.  THE CALLING PROGRAM
   MUST HAVE THE "BLOAD.I" ROUTINE FROM THE UTILITY TOOLKIT
   INCLUDED, AND IT MUST DECLARE "PATHSTRING". *)

VAR
  ERR:INTEGER;

BEGIN
#A
      DSECT         ;SET ASIDE 4K OF HIGH
TABLE EQU $B000     ;MEMORY FOR THE SHAPE
      DS  $1000     ;TABLE.
      DEND
      LDA #>TABLE   ;LSB OF SHAPE TABLE ADDRESS.
      STA $E8
      LDA #<TABLE   ;MSB OF SHAPE TABLE ADDRESS.
      STA $E9
#
  ERR:=BLOAD(SHAPETABLE,0,-20480);
  SETSHAPES:=ERR;
END;

(*====================================*)

PROCEDURE SETRSC(ROT,SCALE,COLOR:INTEGER);

(* SET ROTATION, SCALE AND COLOR OF NEXTSHAPE. *)

BEGIN
#A
      STX _T        ;SAVE X REGISTER.
      LDY #5
      LDA (_SP),Y   ;COLOR
      STA _T+1
      INY
      INY
      LDA (_SP),Y   ;SCALE
      STA $E7
      INY
      INY
      LDA (_SP),Y   ;ROT
      STA $F9
      LDX _T+1
      JSR $F6F0     ;SET COLOR.
      LDX _T        ;RESTORE X REGISTER.
#
END;
```

```
(*====================================*)

PROCEDURE DRAWIT(NUMBER,XPOS,YPOS:INTEGER;X:BOOLEAN);

(* USED BY THE DRAWSHAPE AND XDRAWSHAPE ROUTINES.  THIS PROCEDURE
   DOES THE ACTUAL SHAPE DRAWING. *)

BEGIN
#A
      STX _T        ;SAVE X REGISTER.
      LDY #5
      LDA (_SP),Y   ;X-->(DRAW OR XDRAW?)
      STA _T+1
      INY
      LDA (_SP),Y   ;YPOS
      STA _T+2
      INY
      INY
      LDA (_SP),Y   ;LSB OF XPOS.
      STA _T+3
      INY
      LDA (_SP),Y   ;MSB OF XPOS.
      STA _T+4
      INY
      LDA (_SP),Y   ;SHAPE NUMBER.
      LDY #0
      CLC
      CMP ($E8),Y   ;COMPARE SHAPE NUMBER TO TOTAL NUMBER OF SHAPES.
      BCS ENDSHAPE  ;DON'T DRAW IF THE SHAPE NUMBER IS LARGER
                    ;THAN THE TOTAL NUMBER IN THE TABLE.
      ASL           ;OTHERWISE DOUBLE THE SHAPE NUMBER SO
                    ;IT CAN BE USED AS AN INDEX TO THE
      STA _T+5      ;ADDRESS OF THE SHAPE.
      LDX _T+3      ;LOAD THE REGISTERS
      LDY _T+4      ;WITH THE SCREEN COORDINATES
      LDA _T+2      ;THE SHAPE WILL BE DRAWN AT.
      JSR $F411     ;SET THE CURSOR POSITION.
      LDY _T+5
      LDA ($E8),Y   ;GET LSB OF RELATIVE SHAPE ADDRESS
      STA _T+6
      INY
      LDA ($E8),Y   ;GET MSB OF RELATIVE SHAPE ADDRESS
      STA _T+7
      CLC
      LDA _T+6      ;NOW DO A TWO BYTE ADDITION
      ADC $E8       ;WITH THE LOCATION OF THE
      TAX           ;START OF THE SHAPE TABLE
      LDA _T+7      ;TO GET THE ABSOLUTE
      ADC $E9       ;ADDRESS OF THE SHAPE AND
      TAY           ;PUT IT IN THE X AND Y REGISTERS.
      LDA #0
      CMP _T+1      ;DRAW OR XDRAW?
      BEQ DRAW
      LDA $F9       ;ROT
```

```
      JSR $F65D     ;XDRAW THE SHAPE.
      JMP ENDSHAPE
DRAW  LDA $F9       ;ROT
      JSR $F601     ;DRAW THE SHAPE.
ENDSHAPE LDX _T     ;RESTORE X REGISTER.
#
END;

(*====================================*)

PROCEDURE DRAWSHAPE(NUMBER,XPOS,YPOS:INTEGER);

(* DRAW A SHAPE *)

VAR
  X:BOOLEAN;

BEGIN
  X:=FALSE;
  DRAWIT(NUMBER,XPOS,YPOS,X);
END;

(*====================================*)

PROCEDURE XDRAWSHAPE(NUMBER,XPOS,YPOS:INTEGER);

(* XDRAW A SHAPE *)

VAR
  X:BOOLEAN;

BEGIN
  X:=TRUE;
  DRAWIT(NUMBER,XPOS,YPOS,X);
END;
```

```
#A
 USESHIRES
#

PROGRAM SHAPETEST(INPUT,OUTPUT);

(* TEST OF SHAPE DRAWING ROUTINES *)

TYPE
  PATHSTRING=ARRAY[1..65] OF CHAR;

VAR
  I,X,Y:INTEGER;

#I BLOAD.I
#I HIRES.I
#I SHAPES.I

BEGIN
  I:=SETSHAPES(´ASCII.SHAPES
´);
  HGR;
  SETRSC(0,1,3);
  Y:=10;
  X:=5;
  FOR I:=1 TO 127 DO BEGIN;
    X:=X+20; (* YOU MAY HAVE TO USE A DIFFERENT INCREMENT IN X
                IN ORDER TO SEE YOUR SHAPES PROPERLY. *)
    IF I MOD 14 = 0 THEN BEGIN;
      X:=5;
      Y:=Y+15; (* YOU MAY HAVE TO USE A DIFFERENT INCREMENT IN Y
                  IN ORDER TO SEE YOUR SHAPES PROPERLY. *)
    END;
    DRAWSHAPE(I,X,Y);
  END;
#A
  JSR $FD0C  ;WAIT FOR KEYPRESS
#
  TX;
END.
```

LETTERS

Mark A. Smith from Pensacola, Florida writes:

"I ordered my copy of Kyan Pascal last April 86 so that I could teach myself how to program in Pascal, mostly for the fun of it (and it is fun). I received my copy of Kyan Pascal with the manual right away. But I also had a surprise extra, though I did not know it at the time, for I found a disk filled with KIX files, as well as a large portion of my users manual dedicated to KIX. Though I know nothing of UNIX, I expected a UNIX-like operating environment as you had advertised in the March issue of inCider... Then I received my June issue of inCider and received quite a shock. I saw an advertisement for Kyan Pascal PLUS! Then I saw advertisements for KIX alone as an AppleWorks enhancement. During this time, I had been learning to use KIX and growing more excited the more I learned; but I suddenly realized that when I had order Kyan Pascal for the Apple, what I had really received was Kyan Pascal PLUS!...Did you ship it to me as a preannounced product to test user response??...Most importantly, I would like to know if I am a registered owner with Kyan Software Inc. as legitimate owner of KIX? Will I be allowed to upgrade to KIX version 1.02 if I just send in my original disk with a request?

Dear Mark, in fact your guess is correct. We did ship the early versions of Kyan Pascal with the KIX environment included with the purpose of testing the user response. Also, you are allowed to receive an upgrade to KIX 1.02, as anybody is who has the original source disk. Further, you are also entitled to order Appleworks KIX for the special offer of $20.00 (plus 4.50 for shipping and handling).

Eugene Vamos of La Canada, California, writes:

"...How can I get back issues of Update...Kyan #2, #3, and #4?...

Regrettably, we have run out of most of our previous issues. Many new subscribers want to get their hands on back issues, but we only print enough to send to the subscribers and have few ,if any ,left over.

UPDATE ... KYAN

Kyan Software Inc.
1850 Union Street #183
San Francisco, CA 94123

November/December Issue
Volume 2, Number 1

Apple Edition

WHAT'S NEW?

What We're Doing With the IIGS....

by John R. Fachini

By now I am sure all of you have heard/read/seen lots and lots of things about the Apple IIGS, so I am not going to waste your time telling you about its technical qualities, its great sound and graphics, or its speed. What I will tell you about is what we at Kyan Software, Inc. have been working on with our two GS computers since January.

Kyan has great plans for 1987 and the Apple IIGS is a large part of those plans. The number of products we are planning to release next year will more than double our current product offering. Even better, the product quality will be increasing, but price won't be. We are trying to improve technical support with the introduction of a technical support bulletin board sometime in early spring. We will be posting helpful hints, bug fixes, and other handy items which will save you time and hopefully aggravation as the adjustment from the Apple IIe world to the IIGS world slowly takes its toll on you, the fearless programmer.

There is a lot to say, so let me start by stating something I learned a few weeks ago at the Fall 1986 Apple Developers Conference in San Jose, California. The event was excellent and I had a great time. I did notice, however, there is a great lack of development tools aimed at those programmers who have used the Apple IIe and IIc. I know of more than half a dozen cross compilers for Macintosh to IIGS programs. But there is no Pascal for Apple II people to start using now which will still work when they get their IIGS!? WRONG!

If you want to get a head start on your IIGS application, grab some Kyan toolkits and get to work. Once our GS products are released, you can move your Pascal sources over and re-compile. There you will have a 16-bit version using all of the super features of the GS without any extra waiting or sweating the lack of available GS computers.

This leads me to my second topic....our product list for the IIGS. Beginning in 1987, we will phase in what we believe will be the most powerful development environment and programming tools available for the Apple IIGS. That's a big statement, but we haven't been busting our tails for months to take it slow now.

The introduction schedule looks something like this:

First Quarter, 1987

1. Kyan Pascal/GS with KIX/GS
2. System Utilities (DOS and misc.)
3. Advanced Graphics Toolkit
4. Human Interface Toolkit (MouseGraphics)

Second Quarter, 1987

5. Sound Toolkit (including speech)
6. Assembler/Linker
7. Kyan Pascal with separate linking

Third Quarter, 1987

8. Kyan "C"

Needless to say, there is lots of power in the above list. And that's just for starters (I guess I won't be taking a vacation for a while). Let me clarify what must be some fuzzy points about the above list.

Kyan Pascal plus KIX for the GS will be the first release. It will be our ISO Pascal (with extensions like STRINGS!!!!!!!!), using 32-bit integers, SANE floating point support, a full power 65816 in-line assembler, and the power to let you use all of your memory for program space (automatic segmentation), all memory for dynamic space, and up to 64 K of global variables.

KIX will have a 'new' look. It will have "history" support (use the up and down arrows to scroll through your last 50 typed lines of input), the AppleWorks flashing underline input routine, command scripts (you know, batch files?), "alias" commands, error and standard redirection, input redirection (for talking to KIX from a disk file or a modem), and lots of other great things.

The toolkits are pretty clear. The existing Apple //e and //c MouseGraphics and Mousetext Toolkits will become the Human Interface Toolkit on the GS. It will include code for the 'Mac' look and feel; it will also include user friendly design suggestions with examples.

The sound toolkit is my personal favorite. Save a lot of disk space for your speech files. I guess if things go well, you'll be able to code something like:

```
SPEAK (Vocab[1],2)
```

and hear words come out of Pascal. Amazing.

The Assembler/Linker will let you generate all of the types of modules the GS System Loader supports. The Linker will let you have libraries of routines out of which the linker will copy only those routines you need. That includes routines of Pascal code, assembly code, C code, etc. (since they all compile into 65816 code anyway....)

The second release of Kyan Pascal/GS will be a special version with language enhancements which will let you compile library modules, etc. Life gets interesting at that point.

Most of the people I talk to in the course of my technical support duties ask about a C compiler. Well, it's official. There will be a GS C compiler from Kyan Software. The implementation of C will be a full K&R implementation with extensions and special GS constructs. Watch future issues of **Update...Kyan** for more details.

To give you a peek at some other potential releases, let me mention two products (and write if you want either one or both for the GS and IIe/c):

- A programming editor (context sensitive even!)

- A LISP interpreter/compiler.

Let us know what you think.

As soon as the GS is an established product, we will be starting the Update...Kyan GS version of our bi-monthly newsletter. I have already been warned that I'll be the editor and (so far) only contributing writer. So don't be bashful--send us some code, folks (even if it's not on the GS).

All of this hype and excitement and I still haven't mentioned the Apple IIe and IIc product lines. Well, don't fret gang, we're taking care of you, too. In March there will be a KIX upgrade and a Pascal Version 2.1 release. Pascal will get a faster assembler and strings. If you want more, send me a letter telling me what you want. Speak up! Speak also up about what you want for the GS!

UPDATE

LIB Copyright Information

Because of the speed and stand-alone ability of its machine code, many people use Kyan Pascal to write programs to be distributed in the public domain (PD). There is some confusion as to how a person places their program in the public domain because of the Pascal Library (LIB) file, which is a copyrighted product of Kyan Software, Inc.

If the file is to be distributed on disk (such as one in a user group PD software library), copy the LIB file there as usual, but you must acknowledge Kyan's copyright on the disk label with the following message:

LIB Copyright ©1986 Kyan Software, Inc.

Also insert this message in a prominent location in the documentation.

If the file is to be distributed via a BBS, you may put the LIB file up separately, providing that the above © message is in the documentation to the program.

PASCAL PROGRAMMING

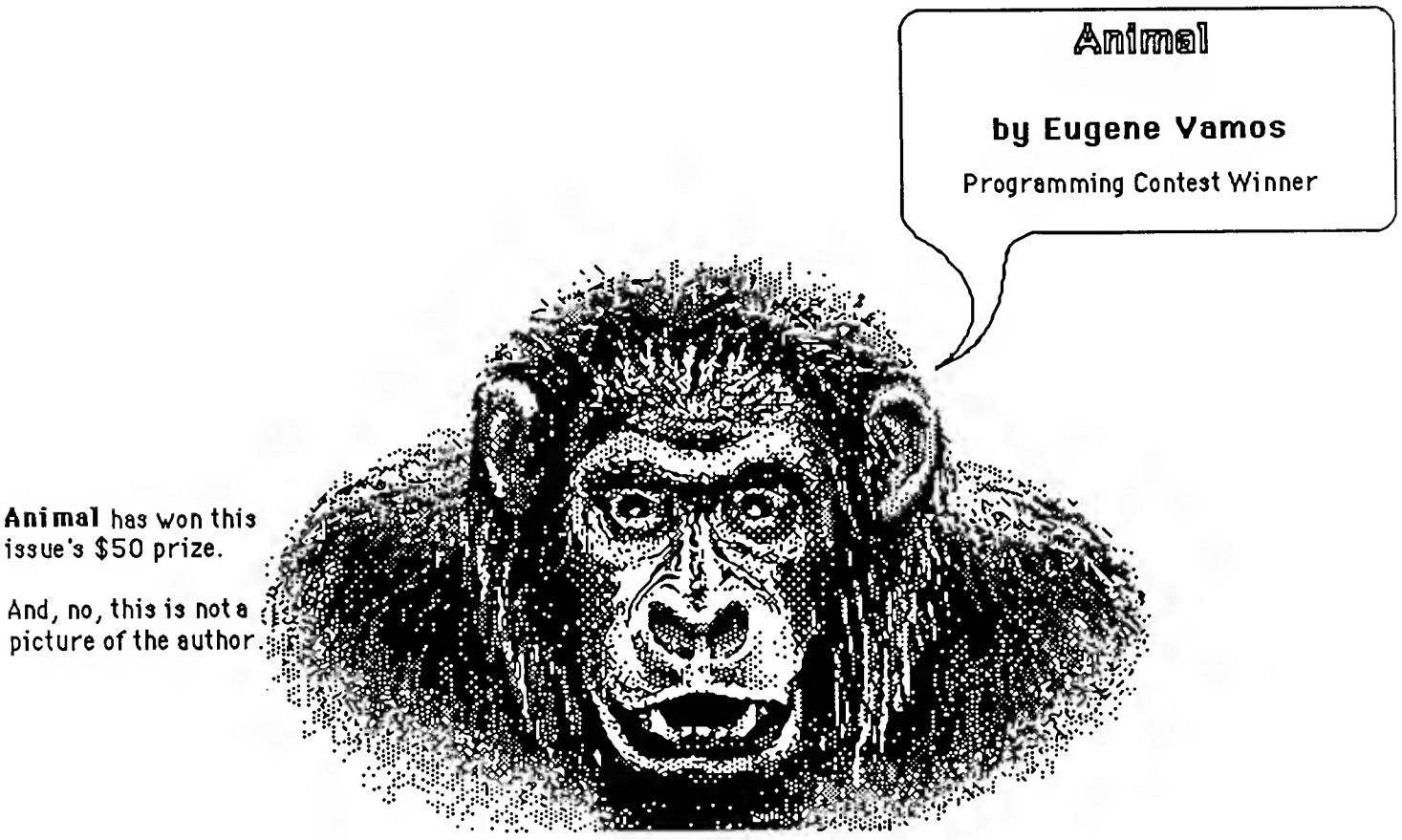
Programming Contest

If you have ever checked out a complete copy of ProDOS, you may have played around with a game on the disk called *Animals*. This month's Programming Contest winning

program, by Eugene Vamos of La Canada, California, is similar to the ProDOS *Animals* BASIC program. Mr. Vamos also wrote a descriptive article to accompany his program; his efforts have been rewarded with fifty dollars, as are all of the Programming Contest winners (Hint, hint! Send in those programs/routines!).

This program demonstrates the portability of Pascal--it can be compiled by any Standard Pascal compiler with no changes whatsoever--it was first written for Atari computers, but also runs when compiled by our Apple and Commodore compilers.

The article follows and the program listing can be found at the end of the newsletter.



What is Animal? you say. Well, Animal is a question and answer game that looks like it has touches of artificial intelligence. The user thinks of an animal (or anything else for that matter) and the program, by means of *yes* and *no* questions, tries to guess it. As one plays the game again and again, it provides better, more accurate guesses as it learns from its mistakes.

Programming Techniques Used

Animal uses a binary tree to store the questions and the guesses. The variable **Toggle** of type **Decision** determines if the program should ask a question or guess at the answer when it arrives at a particular record node. At initialization, only three nodes exist: the root node with the first question, and two children nodes with guesses (Figure One).

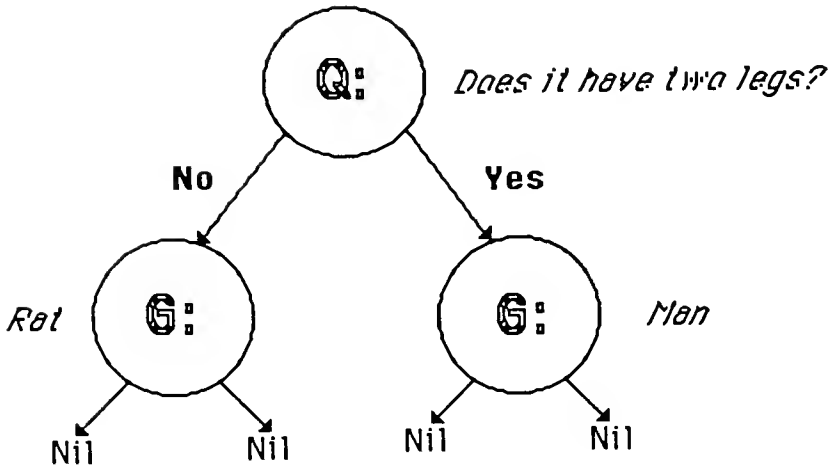


Figure 1.

If the response to a question is *no*, the pointer moves left, but if the response is *yes*, then it goes in the opposite direction, right. the program then checks what kind of node it is and does the appropriate action (to question or to guess).

When the program guesses at the animal and gets it right, it congratulates itself (to the point of being a bad winner). On the other hand, if it's wrong, the sore loser asks for the right answer and for a question that, if answered affirmatively, would describe the animal. the program stores this new information by creating two new nodes and attaching them to the tree. the way the program does this is better illustrated by showing, not telling (Mrs. Stinson--English 3AB5). Taking the sample tree from above, a trial run would look like this *(program output appears in regular typeface; user input appears in bold)*:

Q: Does it have two feet? (Y/N)
Y
G: Is it a Man?
N
What was it then?
Kangaroo
What question should I have asked?
Does it bounce?

The tree would then look like Figure 2.

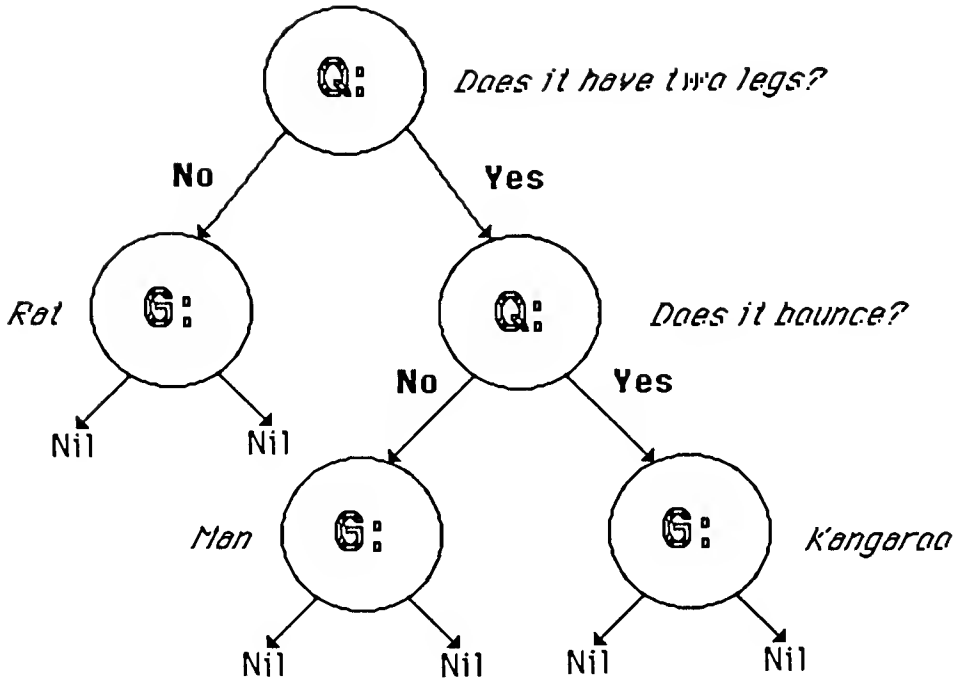
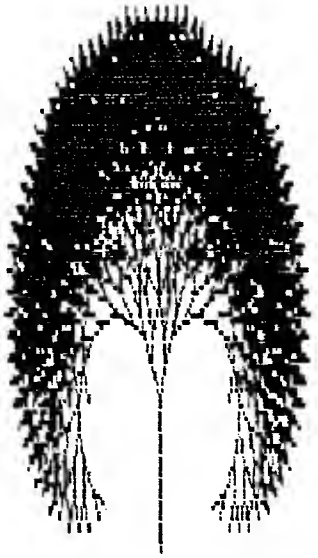


Figure 2.

This is a 'bare bones' program, and other features could be added, such as crash protection from bad input and a save to disk procedure *(that way you could save a huge disk database of animals from alligator to zebra because saving all the animals and questions in memory may take up too much space and you may overflow the available memory--Ed.).* In time, one could create a giant binary sequoia that would amaze and impress *one's friends (and make the Fractal Tree, which appears on the next page, look like a mere wimpy sapling--Ed.).*

Fractal Tree



This Pascal program uses fractal mathematics (which are often used with computer graphics) in the recursive procedure **Split** to make the branches of a tree in graphics mode eight. This program is a derivative of the **Fractal** module from TDI Software's **Modula-2/ST GemDemo** program.

One might think that it would be extremely difficult to convert the program from Modula-2 on an Atari 1040ST, which uses a 68000 microprocessor running at 8 Mhz, *down* to Pascal on an Apple //e, which uses a 1Mhz 6502 (usually, it is done the other way--Pascal programs are converted *up* to Modula-2), but it was fairly easy. Except for forgetful 'cockpit errors' such as forgetting to include the graphics file, the program compiled and then ran (successfully, even!) without changes from the first translation of the program (with me, a first-time compilation *and* successful execution is about as common as a complete solar eclipse coinciding with a Boston Red Sox World Series victory). Now that's Pascal <--> Modula-2 compatibility at its best (thank you, Professor Wirth!).

And, lest I forget, you had better have something to do while you run this program (give your dog a bath, make dinner, etc); as you may have guessed if you know anything about recursion, floating point, and a 1Mhz CPU, *its gonna take a while, folks!* What took the 68000 machine thirty seconds to do takes your 6502 machine about twenty minutes; it will be interesting to see how long it takes the 65816-powered, 2.8Mhz IIGS to draw the whole tree. If anybody is brave enough (and has an extra day or so), I'd like to know the time it takes BASIC to do this program.

Watching the fractal tree draw is about as much fun as watching paint dry, but the program uses some pretty slick techniques (fractals, recursion) and the end graphics result is fairly decent.

The program listing appears at the end of the newsletter.

LETTERS

Is it possible to read the command line from inside a Pascal program?

Chris Keller
[75776,2400]
CompuServe

*Assuming that you want to read the KIX command line and then perform commands (such as **LS**, **CP**, etc.) ,no--the KIX environment is separate from your program. You can, however, read your own command line using the **Parse Line** routine from the System Utilities Toolkit and then perform one of the many file management commands available in the ProDOS library of that same toolkit.*

I don't understand how to use the **Address** and **Pointer** commands to get a BASIC-type PEEK and POKE.

Clarence J. Arrowsmith
[73307,3152]
CompuServe

To do a PEEK or POKE, you don't need to use ***Address*** *at all, only* ***Pointer****. First, declare a number that you will use to PEEK and/or POKE the memory location(s):*

```
VAR
        My_Loc : ^Integer;
    Peeker, Poker : Integer;
```

Next, you must assign your variable to the appropriate memory location:

```
    My_Loc := Pointer(1234);
(*  You would, of course, use the proper memory location in place of 1234  *)
```

Later in your program, you can perform a PEEK with:

```
    Peeker := My_Loc^;  (*  Peeker will now hold the value in address 1234  *)
```

...or a POKE with:

```
    Poker := 255;  (*  255 is the number you are going to POKE into 1234  *)
    My_Loc^ := Poker;  (*  This performs the POKE  *)
```

Is there ANY way to get around input type clashes in Pascal? The most annoying thing in the world is to be prompted for a number, enter a letter by mistake, and have the program come crashing down into your lap.

Chris Copeland
Monroe, Connecticut

ISO Pascal ***requires*** *that when a program encounters an error such as a type conflict or an out-of-range array index, it must come screeching to a halt and report the 'fatal error' to the user. To get around this, design some kind of 'bullet-proof' input routines; for example, have separate procedures such as* ***Read_Int, Read_Real****, etc.*

NOT-SO-SUBTLE HINT: What a great idea for our ongoing programming contest! Remember that each issue, one person who writes a nice routine or program will be chosen to receive fifty dollars and have his or her code published on these very pages.

Here are my submissions to **Update...Kyan** for this month:

First a procedure...

```
Procedure ExitProgram;

{ this procedure brings a program to  }
{ an orderly halt by calling the      }
{_quit routine in StdLib.s            }
{ ProDOS's quit call will close all   }
{ files.                              }

Begin
#A
```

```
        JMP    _QUIT
#
End;
```

And a Control-Reset patch. This patch to **StdLib.s** will make Control-Reset either quit to KIX or ProDOS or it can make a program restart (be careful with open files if the program is restarted). Put this patch at the beginning of **StdLib.s**.

```
;-----------------------------------------------
; Control-Reset to _quit (or _start) patch
;-----------------------------------------------
        LDA   #>_QUIT       ;change "_QUIT" to "_START" to...
        STA   $3F2          ;make program restart...
        LDA   #<_QUIT       ;instead of quitting
        STA   $3F3
        EOR   #$A5
        STA   $3F4
;-----------------------------------------------
```

Hope this code proves useful.

Jim Luther
Overland Park, Kansas

Many readers will doubtless find these routines very handy.

How can I get some detailed information on the runtime library (LIB) and/or macro library (StdLib.s)?

(various users)

This is one of the more common questions we receive here at Kyan. The Code Optimizer Toolkit contains full source code to the runtime library and a special optimized macro library.

```
Program Animal(Input,Output);
 Const Blank='                         ';
                (* Twenty Five Spaces *)

 Type  String=Array[1..25] of Char;
       Decision=(Answer,Ask);
       Pointer=^RecPtr;
       RecPtr=Record
                Left,Right:Pointer;
                Question:String;
                Animal:String;
                Toggle:Decision
               End;

 Var Root,Mover,Tree:Pointer;
     Entry:Char;
     Temp,Name,Quest:String;
     Part,Whole:Boolean;


 Procedure Init;

  Begin
   New(Tree);      (* Set up Root node*)
   Root:=Tree;
   Mover:=Tree;
   Tree^.Toggle:=Ask;
   Tree^.Question:='Does it have two feet     ';
   Tree^.Animal:=Blank;

   New(Tree^.Right);   (*Set Up Right Branch*)
   Tree^.Right^.Toggle:=Answer;
   Tree^.Right^.Question:=Blank;
   Tree^.Right^.Animal:='Man                          ';
   Tree^.Right^.Left:=Nil;
   Tree^.Right^.Right:=Nil;

   New(Tree^.Left);   (*Set up Left Branch*)
   Tree^.Left^.Toggle:=Answer;
   Tree^.Left^.Question:=Blank;
   Tree^.Left^.Animal:='Rat                           ';
   Tree^.Left^.Left:=Nil;
   Tree^.Left^.Right:=Nil;
   Tree:=Root;
   Mover:=Root;
 End;
```

```
Procedure AskQuestion;

 Var Response:Char;

 Begin
  Writeln('Q: ',Mover^.Question,' (Y/N)');
  Readln(Response);
  Case Response of
   'Y':Mover:=Mover^.Right; (* Move to the left *)
   'N':Mover:=Mover^.Left   (* Move to the right *)
  End;
 End;


Procedure SayGuess;

 Var Response:Char;

 Begin
  Part:=False;
  Writeln('G: Is it a ',Mover^.Animal);
  Readln(Response);
  Case Response of
   'Y','y':Writeln('Ha, Ha , I guessed it. I win!');

   'N','n':Begin
             Writeln('What was it then?');
             Readln(Name);
             Writeln('What question should I have asked?');
             Readln(Quest);
             Tree:=Mover;
             (*Move the pointer to the action*)
             Temp:=Tree^.Animal;


             New(Tree^.Left);(* Create Two new nodes *)
             New(Tree^.Right);

             Tree^.Left^.Toggle:=Answer;
                              (* Move contents of *)
Tree^.Left^.Question:=Blank; (* current node to the *)
             Tree^.Left^.Animal:=Temp; (* new left node *)
             Tree^.Left^.Left:=Nil;
             Tree^.Left^.Right:=Nil;
```

```
Tree^.Right^.Toggle:=Answer;  (* Set up new right *)
Tree^.Right^.Question:=Blank; (* node with new *)
             Tree^.Right^.Animal:=Name;
                           (* animal guess *)
             Tree^.Right^.Left:=Nil;
             Tree^.Right^.Right:=Nil;
             Tree^.Toggle:=Ask;
                           (* Place new question *)
             Tree^.Question:=Quest;
                           (* at current node *)
             Tree^.Animal:=Blank;
             Temp:=Blank;
             Tree:=Root;       (* Move pointers *)
             Mover:=Root;      (* back to the root *)
            End
         End;
      End;


 Begin
  Init;
  Mover:=Root;
  Writeln('Welcome to Animal, the game.');
  Writeln('  In the game, the computer tries to guess the');
  Writeln('animal you have thought of by asking Yes (Y)');
  Writeln('and No (N) questions.');
  Writeln('  When and if the computer guesses wrong, it');
  Writeln('will try to learn from its mistakes and will');
  Writeln('ask you some information about your animal.');
  Writeln;
  Writeln;
  Writeln('If you are ready, here we go!!!');
  Whole:=True;
  Part:=True;
  While Whole=True Do
   Begin
    While Part=True Do
     Begin
      Case Mover^.Toggle of
       Ask:AskQuestion;
       Answer:SayGuess
      End;
     End;
    Writeln('Do you want to play again(Y/N)');
    Readln(Entry);
    If Entry='N' Then Whole:=False
     Else Part:=True;
   End;
  End.
```

```
#A
_UsesHires
#

PROGRAM Fractal_Tree;

CONST
              xScaler = 1.0;
              yScaler = 1.0;
              Color   = 6;
            MaxLevel  = 13;
          SplitAngle  = 20.0;
     SplitProportion  = 0.22;

  VAR
     xMax, yMax      : Real;
     dx, dy,
     mx, my,
     CosPhi, SinPhi  : Real;
     Tab             : ARRAY[0..20] OF Real;
     Loop            : Integer;

#i /Hard1/Pascal/HiRes.i
(*  This is just the pathname for our system;
    you should use whichever pathname your system
    needs to include "HiRes.i"  *)

PROCEDURE Split(sx,sy,ex,ey : Real; Level : Integer);

  VAR
     MidX, MidY,
     OldLen, NewLen,
     CosTheta, SinTheta : Real;
     i : Integer;

BEGIN (* Split *)
  MidX := (sx + ex) / 2.0;
  MidY := (sy + ey) / 2.0;
  dx := ex - sx;
  dy := ey - sy;
  OldLen := Tab[Level-1] * 150.0;
  Newlen := Tab[Level] * 150.0;
  CosTheta := dx / OldLen;
  SinTheta := dy / OldLen;

  Draw(Trunc(sx),Trunc(yMax - sy),Trunc(MidX),Trunc(yMax - MidY),Color);
```

```
  IF Level < MaxLevel THEN
    BEGIN
Split(MidX,MidY,NewLen * (CosTheta * CosPhi - SinTheta * SinPhi) + Midx,
                NewLen * (SinTheta * CosPhi + CosTheta * SinPhi) + MidY,
Level + 1);
Split(MidX,MidY,NewLen * (CosTheta * CosPhi + SinTheta * SinPhi) + MidX,
                Newlen * (SinTheta * CosPhi - CosTheta * SinPhi) + MidY,
Level + 1)
    END (* IF/THEN *)
 END; (* Split *)


BEGIN (* main program module *)

  HGr;

  xMax := 279.0;
  yMax := 149.0;
  Tab[0] := 3.0;
  CosPhi := 0.93969262;  (* Cos(SplitAngle) *)
  SinPhi := 0.342020143; (* Sin(SplitAngle) *)
  FOR Loop := 1 TO 20 DO Tab[Loop] := SplitProportion * Tab[0];
  mx := 279 DIV 2;
  my := (149 DIV 2) - 50;
  Split(mx,0.0,mx,my,1);

  Tx

END. (* main program module *)
```

UPDATE ... KYAN

Kyan Software Inc.
1850 Union Street #183
San Francisco, CA 94123

March/April Issue
Volume 2, Number 3

Telephone: (415) 626-2080 CompuServe ID: 73225,450 MCI ID: 298-0892
EasyLink: 62921785 Telex: 989113 KYAN SFO

Apple Edition

(Editor: David J. Rudolph)

UPDATE

Latest Software Versions

Kyan Pascal:	2.02A
Kyan Pascal PLUS	
Disk 1:	2.02A
Disk 2:	1.02
System Utilities Toolkit, MouseText Toolkit, Advanced Graphics Toolkit, TurtleGraphics Toolkit:	1.00
Code Optimizer Toolkit:	1.01
KIX (AppleWorks version):	1.01

All software remains unchanged since the last issue of UPDATE...KYAN, with the exception of the Code Optimizer.

From the Editor

I would like to first take time to thank the people who responded to my request in sending in programs that were developed using Kyan Pascal. The range of programs we received went from those that help Audiologists ('specialists in the rehabilitation of persons with hearing loss') with their patients to educational software. The authors, one and all, developed great packages in their respective fields of software development. Thank you again for participating.

Back to business though, this issue is going to deal with recursion. This seems to be a neglected topic that has been around for years, yet computer scientists have recently (within the past twenty years) begun to utilize its inherent power. The first program is (you guessed it) "Towers of Hanoi," the ancient game that has regained popularity due to computer scientists implementing it on various machines. Secondly, an excellent article by Robert Oyung on searching techniques covers three standard methods of searching: BubbleSort, MergeSort, and QuickSort.

PASCAL PROGRAMMING

Recursion

Recursion is an old field of mathematics that is finding many applications in computer science, from languages driven by recursion, such as LISP, to faster ways of sorting lists, to searching binary trees. Indeed, recursion is a very handy and powerful tool and is easily implemented in Pascal, whereas in BASIC and FORTRAN it must be kludged since it is not inherent in the original design of the language. Recursion, simply stated, is a way of having a procedure call itself in order to accomplish a task. By having the procedure call itself, it is in fact looping.

The classic game 'Towers of Hanoi' demonstrates this facet of recursive looping. In this game we start with three pegs, or towers. Each peg normally has five disks of varying sizes (we will use three to simplify the example). The disks are stacked from small to large. (See Figure 1.)

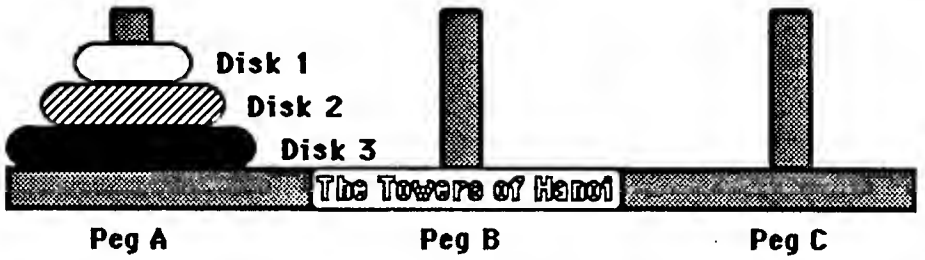


Figure 1.

The object of Towers of Hanoi is to move the stack of disks to another peg without ever placing a larger disk on top of a smaller disk. For example, you can start by moving disk 1 to peg B; then, disk 2 to peg C; then, disk 1 to peg C, etc., etc., etc...

In this example, we start off with three disks on peg A; then, two moves later, we have two disks on peg C. Finally, all the disks end up on peg B. After some experimentation you will discover that whenever you move a larger disk, you must move the smaller one(s) a total of N^2-1 times. With three disks you must move the disks eight times to complete the puzzle with the least number of moves (3^2-1=8). In closing, look over the Hanoi program carefully to understand how recursive loops are made to move the disks from peg to peg. Finally, notice how the state of a variable is saved and recalled during iterations of the recursive loop. I hope you find the 'Towers of Hanoi' an enjoyable and educational game.

PROGRAM Hanoi appears in the LISTINGS section at the end of the newsletter.

Taking a Look at Sorts

by Robert Oyung

An entry in a list is very easy to find, if the list is sorted. Our lives have been made much easier by sorted lists. Names in a phone or address book are sorted, catalog numbers in a library are sorted, and house numbers are sorted. Can you imagine trying to find 3612 Pine Street. without knowing that 3612 is between 3600 and 3700? No thank you. Sorted lists are much easier to deal with.

In order to sort a list, we must do two things:

1) Compare one element with another, and
2) Switch them if necessary.

Consider a very simple list of two numbers; for example, the list: 6 1. The first step is to compare the two elements: 6 is larger than 1. The second step is to switch them if necessary: since 6 is larger than 1, it is necessary to switch the elements. So our list becomes: 1 6. Easy.

Now, let's take a look at the options we have for sorting:

1. The easy way: Bubblesort is one of the simplest sorting algorithms; it is also one of the slowest. It starts at the beginning of the list comparing each element with the next, exchanging them so that the smaller value is on the left and the larger value is on the right. This has the effect of moving the largest number to the end of the list. The procedure continues by moving the second largest element up behind the largest and so on until the list is sorted. Simple, but slow.

2. Divide and conquer: Mergesort embodies the idea of dividing a problem into smaller pieces and working on them first. Mergesort is a recursive procedure that continually divides a list into two parts. The first part is sorted, the second part is sorted, and then the two lists are combined into one sorted list. Getting complicated, but faster!

3. The fastest: Quicksort. To better understand how quicksort works, lets use an example and sort the list:

4 6 8 1 2 3 9 5 7

The middle element is 2. This will be our "pivot" point. Start from the left and find a number greater than or equal to the pivot (2); that number is 4. Then start from the right and find a number less than or equal to 2; that number is 2.

4 6 8 1 2 3 9 5 7

Switch those two numbers.

2 6 8 1 4 3 9 5 7

Now look at the elements between the two numbers we just switched.

6 8 1

Start from the left and find a number greater than or equal to 2 (the pivot). That number is 6. Then start from the right and find a number less than or equal to 2; that number is 1; switch them.

1 8 6

There is only one element between 1 and 6 (8) so all the elements to the right of 8 (including 8) are put in another pile.

2 1 / 8 6 4 3 9 5 7

We sort the new pile the same way.

8 6 4 3 9 5 7

3 is the middle element. The two numbers to switch are 8 and 3.

3 6 4 8 9 5 7

Look at the numbers between the two we just switched.

6 4

We can find a number greater than 3 (the pivot) but we can't find one smaller, so all the elements to the right of 6 (including 6) are put in another pile.

2 1 /3/ 6 4 8 9 5 7

Sort the new pile.

6 4 8 9 5 7

The middle element is 8. Make 2 switches.

6 4 7 9 5 8
6 4 7 5 . 9 8

Sort the new pile.

2 1 /3/ 6 4 7 5 /9 8

8 9

Go back and sort the new pile with the 6. 4 is the pivot.

4 6 7 5

Sort the new pile.

4 / 6 7 5

The middle element is 7.

6 5 7

The list becomes: 2 1 / 3 / 4 / 6 5 / 7 8 9

Now we only have to sort two piles (the pile with the 2 and the pile with the 6) to completely sort the list. In the LISTINGS section of the newsletter, you will find a demo program with the three sorts discussed above. Try them and compare their speeds.

PROGRAM SortDemo appears in the LISTINGS section of the newsletter.

LETTERS

We recieved a letter from Mark A. Smith in Pensacola, Florida,that offers some suggestions on how to use Kyan Pascal with a RAM card. Since the price of RAM has become very reasonable, many users are now buying these cards from various manufacturers and are using them as RAMdisks. Mark's letter describes the process of using a product from a specific manufacturer, Checkmate, but the process will generally be the same no matter what company you purchase your RAM card from..

Mark writes:

An Apple //c with only one drive is very tedious when applications require disk swapping. (For example, Pascal programming files and KIX commands). Here are just a few simple steps of preparation, and then a few tips on using the setup. This method should work fine on the Apple //e with a MultiRAM RGB as well, since Checkmate is famous for its real product compatability.

Preparation:

A. Format a ProDOS disk.

B. Copy files to the front side of your disk in this order:

1. ProDOS

2. MRAM.SYSTEM -- This file is located on the back of the MultiRAM utilites Disk 4.5, included with purchase. It creates the /MRAM disk on the expansion board, thus it needs to be the first system file.

3. LOADMRAM.SYSTEM -- This file, on back of MultiRAM Util., is run to load files from any ProDOS disk into /MRAM, disk after disk, until you are done.

4. Copy Kyan.Pascal files -- I recommend: KIX.SYSTEM, STDLIB.S, LIB, ED, PC, AS, and any include files that you use regularly.

Use of /MRAM RAMdisk:

A. Boot up the front side of the disk you prepared. Here is what happens:

1. ProDOS boots up and runs MRAM.SYSTEM

2. MRAM.SYSTEM creates the RAMdisk (/MRAM) on the expansion card, and then runs LOADMRAM.SYSTEM.

3. LOADMRAM.SYSTEM provides a menu that allows you to copy ALL files from the disk that is currently in the drive into the RAMdisk.

B. Use LOADMRAM.SYSTEM to copy you files on the front side of your disk. When that task is done, flip to the reverse side and copy all of the KIX files to /MRAM. {For those who are familar with Applesoft, you might wish to use the BASIC program COPY.ALL found on the MultiRAM disk. It can be customized to the exact files on your disk(s) you wish to copy to /MRAM.}

C. Exit the menu and, at the ProDOS quit routine, enter:

/MRAM/KIX.SYSTEM <Return>

D. **WHAM!!** No more disk swapping, or waiting for your Apple to access the disk! It all occurs at the "speed of light" inside your computer. Well, at least at the speed of your CPU (much faster than any mechanical drive)! And you have enough space to develop, large sophisticated programs.

TIPS:

A. /MRAM is temporary! It disappears when you shut the power off! If you inadvertantly reset, /MRAM is only disconnected. Follow the instructions in your manual from Checkmate to re-install it.

B. Remember that ProDOS limits the number of files in a volume to 51. You might use a directory for KIX files (a la BIN), or for you program files. COPY II PLUS comes in handy here.

C. Keep a separate "/PROGRAMS" (volume name of your choice) disk in your drive at all times, and regularly save your program source code to theis disk for storage.

I hope this is simple enough. I know some Kyan Pascal users may be a bit discouraged by the relentless amount of disk swapping required for the KIX environment with a single drive. As you can see, this is not necessary. A RAMdisk makes life much easier, and saves a lot of time in the long run.

LISTINGS

The Towers of Hanoi

```
PROGRAM Hanoi;
VAR
  q2,q3,q4,t:CHAR;
  i,p:INTEGER;

   PROCEDURE TOWER(n:INTEGER);
   BEGIN
    IF n > 0 THEN
         BEGIN
   t:=q3; q3:=q4; q4:=t;
   Tower(n-1);
   WRITELN('Move ring',n:2,' from peg ',q2,' to peg ',q4);
   t:=q2; q2:=q3; q3:=q4; q4:=t;
   Tower(n-1);
   t:=q2; q2:=q4; q4:=t
          END
   END;

BEGIN { Main }
 WRITELN(CHR(12)); { clear screen }
 q2:='A'; q3:='B'; q4:='C';
 Writeln('****************** Towers of Hanoi **************');
 WRITELN; WRITELN;
 WRITE('How many disks would you like to move? ');
 READLN(p);
 Tower(p);
END.  { Main }
```

Taking a Look at Sorts

```
Program SortDemo (input,output);

Const  Max = 50;

Type  ArrayType = array [1..max] of integer;

Var
    i: integer;
    original, Sorted : ArrayType;


{*********************************************************}

Procedure BubbleSort(Var list:ArrayType;start,finish:integer;
                Var comp:integer);

{ This procedure sorts a list by repeatedly finding the
largest value and then moving it to the end of the list. }
```

```
Var i,j,temp:integer;

Begin
    for i:= finish-1 down to start do
        for j:= start to i do
      Begin
        { See if the next element is smaller }

                    comp := comp+1;
          if (list [j] > list[j+1] ) then
          Begin           { Switch the values }
            temp := list[j];
            list[j] := list [j+1];
                 list[j+1] := temp;
          end;  { if }
         end;  { for }
  end; { procedure BubbleSort }

{*************************************************************}

Procedure MergeSort (Var list: ArrayType; left, right:
integer;
                Var comp:integer);

{ This procedure divides the list in half and sorts them }

Var
    middle, i:integer;
    tempList:ArrayType;

{*************************************************************}

Procedure Merge (source: ArrayType;  Var destination:
ArrayType;
          index1, bound1, index2, bound2: integer);

{ This procedure merges two sorted lists (in source) into one
sorted list (in destination) }

Var
    current, i:integer;
Begin
    current :=  index1;
    while (index1<= bound1) and (index2<=bound2) do
    begin
{ smallest value is in first list }
          comp:=comp+1;
           if (source[index] < source[index]) then
          Begin
       destination[current] := source[index1];
       index1 := index1+1;
       current := current+1;
    end  else
```

```
    Begin
{ smallest value is in the second list. }
       destination[current] := source[index2];
       index2 := index2+1;
       current := current+1;
    end;
end;

for i:= index1 to bound1 do
Begin
    destination[current] := source[i];
    current := current+1;
end;
for i := index2 to bound2 do
Begin
    destination[current] := source[i];
    current := current +1;
end;
end; { Procedure Merge }

Begin { MergeSort }
    if (left < right) then begin
    midde := (left+right) div 2;
    for i := left to right do
       tempList [i] := list [i];
       MergeSort(tempList,left,middle,comp);  { Sort left half
}
       MergeSort(tempList,middle+1,right,comp); { Sort right
half }
    { Merge the left and right halves to together }
         MergeSort(tempList,list,left,middle,middle+1,right);
    end;
end;  { Procedure MergeSort }


{*********************************************************}

Procedure QuickSort (Var data:ArrayType;
start,finish:integer;
                        Var comp:integer);

{ This procedure uses the recursive quick sort to sort a list
of numbers }
Var
     value,left,right,temp,middle,pivot: integer;

Begin
     left := start;
     right := finish;
     middle := (start+finish) div 2;     { Find the middle of
the list }
     pivot := data[middle];          { Find the value there }
     repeat
        While(data[left]<pivot) do
```

```
        Begin
          left := left+1;      { Find a value large than the
pivot on the left side }
          comp := comp+1;
        end;
        While (pivot<data[right])do
         Begin
          right := right-1;          { Find a value smaller
than the pivot on the right side }
          comp := comp +1;
        end;
        if (left <= right) then
        Begin
          temp := data[left];        { Make sure the two
values are on different sides of the }
          data[left] := data[right];     { pivot and then
switch them }
          data[right] := temp;
          left := left+1;
          right := right -1;
        end;
           until (right<=left);

     if (start<right) then           { Recursively sort the
right side }
        QuickSort(data,start,right,comp);

     if (left<finish) then           { Recursively sort the
left side }
        Quicksort(data,left,finish,comp);
end;  { Procedure QuickSort }

Procedure PrintList (list:ArrayType);

Var
     i:integer;

Begin
     For i := 1 to MAX do
        write (list[i],'  ');
end;

{************************************************************}

Procedure GetKey;
{ wait of the user to press RETURN }

Begin
     write ('Press RETURN to continue');
     readln;
end;

{************************************************************}
```

```
Procedure ShowSort(Sorted:ArrayType; i:integer);

Var
     comp:integer;

     Begin
     PrintList(Sorted);
     writeln;
     comp:=0;
     writeln('***Sorting...');

     case i of
        1 : BubbleSort(Sorted,1,MAX,comp);
        2 : MergeSort (Sorted,1,MAX,comp);
        3 : QuickSort (Sorted,1,MAX,comp)
     end;

     PrintList(Sorted);
     writeln;
     writeln('Comparisons: ',comp);
end;

Begin { Main Program }
     for i := MAX downto 1 do  { Create a decrementing list }
        Original[i] := i;
     for i := 1 to 3 do begin
        Sorted := Original;
        ShowSort(Sorted,i)
     end;
end.
```

KIX Commands

Command Syntax	Description
Directory Management Commands	
CD_*pathname*	Change Working Directory
LS_*pathname*	List Directories and Files
MKDIR_*pathname*	Make a New Director
PWD	Print Working Directory
RMDIR_*pathname*	Delete a Directory
File Management Commands	
CAT_*pathname*	List the Contents of a File
CHMOD_*pathname*	Change Protection Status
CP_*source*_*destination*	Copy a File
LPR_*pathname*	Print a File
MV_*old.name*_*new.name*	Move or Rename a File
RM_*pathname*	Delete a File or Directory
Volume Management Commands	
CPV_(s,d)_(s,d)	Copy a Volume
FORMAT_(s,d)_*/name*	Format a Volume
MVV_(s,d)_*/name*	Rename a Volume
Special KIX Commands	
CMP_*file.1*_*file.2*	Compare Two Files
CMP_(ss,sd)_(ds,dd)	Compare Two Volumes
FIND_*directory*_*-filename*	Locate a File in a Directory
GREP_*string*_*pathname*	Locate a String in a File(s)
SDIFF_*file.1*_*file.2*	Compare Two Text Files
Abbreviations and Wildcards	
. (Abbreviation)	Working Directory
.. (Abbreviation)	Parent Directory
? (Wildcard)	Single Character
* (Wildcard)	Character String
ECHO_*pathname*	Show Wildcard Pathnames
Firmware Calls	
C40	Disable 80 Column Card
C80	Enable 80 Column Card
Date_*yymmddhhmm*	Set/Read System Date/Time
SD	Print Screen Contents
Utility Commands	
CFG	Configure KIX
INSTALL	Install KIX
KIX	KIX Command Summary
QUIT	Exit KIX lo ProDOS

kyan software
1850 union street #183
san francisco, california 94123

First Class Mail

KIX

The Hard Disk Manager

kyan SOFTWARE

Kyan Software Inc.
1850 Union Street, #183
San Francisco, CA 94123

(415) 626-2080

KIX™

"The Hard Disk Manager"

KIX is a powerful operating shell and disk manager designed for power users of the Apple II. It makes ProDOS a more useful operating system and saves you a lot of wasted time.

KIX eliminates the need for the cumbersome ProDOS Filer. From the system prompt, you can manage files; work with directories; format disks; or copy volumes. But that's not all! **KIX** expands the functionality of ProDOS and lets you merge and compare files; print directories; search a disk for key words or filenames; write- or read-protect sensitive files; date-stamp files (without a clock card); dump the screen to a printer; and much, much more. And KIX commands can be called from inside AppleWorks™!

KIX gives you the power to:

- Control Directories: (print, create, delete or change directories);
- Manipulate Files: (copy, move, delete, rename, print or change protection status of files);
- Work with Volumes: (copy, delete, or rename volumes; format disks);
- Compare Files and Volumes: (compare and list differences between multiple files or volumes);
- Search Files and Directories: (search directories for files and/or search files for character strings);
- Make Hardware/Firmware Calls: (set system time/date, 40/80 column output, etc.).

KIX also contains a MENU program for launching application programs. It also supports wildcards and redirection of output to printers and other devices.

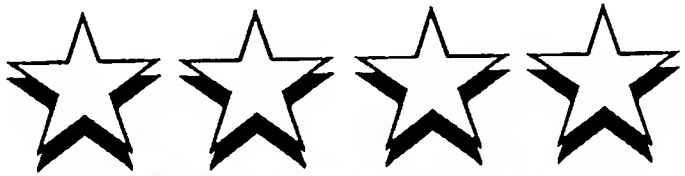
KIX requires an Apple // with at least 64K of memory and a hard disk or Unidisk 3.5. It is compatible with AppleWorks and most other ProDOS-based software. It resides on disk or in RAM and works with most memory expansion cards.

What is KIX?

KIX is a ProDOS-based operating environment for the Apple II. It is modeled after the user environment found in the UNIX™ operating system, and provides the user with many of the utilities found only on large, multi-user UNIX systems.

KIX is not a new operating system. When you examine the **KIX** disk, you'll find ProDOS listed in the directory. And, when you boot the disk, you'll see the ProDOS copyright screen, notifying you that ProDOS is being loaded into memory.

KIX is a new interface for ProDOS. When **KIX** is loaded, you will see the **KIX** prompt (%) instead of the ProdDOS promt (>). When you enter a command, **KIX** interprets the command, calls the necessary utilities or application programs, and works with ProDOS to carry out your instructions.



RATED 4 STARS by InCider Magazine.

"KIX gives ProDOS users access to many features of more powerful operating systems such as MS-DOS and UNIX. If your Apple II system includes a high-capacity disk drive (UniDisk 3.5 or hard disk), you should buy KIX."

"In all, KIX contains more than two dozen commands and utilities that are a welcome sight to anyone who has to manage files on a hard disk."

"Overall, KIX provides the ProDOS control Apple left out. Serious users shouldn't be without it."

... InCider Magazine, January, 1987

How Does KIX Work?

KIX consists of more than 25 UNIX-like commands or utilities. To call these utilities, you use the following command syntax:

Utility_-Options_Arguments_>n

The Utility portion of the **KIX** command syntax is the abbreviated name of the command you want to invoke (e.g., "LS" to list a directory, "CP" to copy a file). These utilities are described later.

The Options portion of the command syntax instructs the utility program to perform the command in a specific way. Many **KIX** commands have options associated with them. For example, the "LS" command provides options for standard or extended directories, alphabetized file names, listing of file protection status, and more. By using options, you can customize the output of the **KIX** utility to meet specific needs.

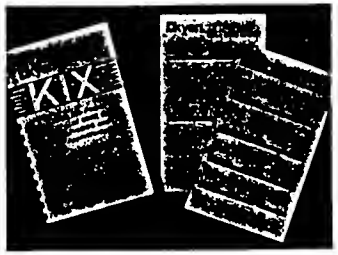
The Argument portion of the **KIX** command syntax is a pathname. The argument defines the file or directory name(s) associated with the command. For example, if a command is given to delete files on a disk, the argument would specify the pathname of the file or files (more than one can be specified) to be deleted. If a Move command is given, two arguments would be provided; the first would indicate the pathname of the file to be moved, and the second where the file is to be moved.

KIX also supports Redirection which allows you to direct the output of a command to a specified slot number "n" (e.g., printer, modem, or other peripheral) or to a pathname (i.e., save the output as a text file). The redirection option is especially useful for such things as printing a hard copy of a disk directory, merging text files, and printing file listings.

KIX is available at local retail stores or directly from Kyan Software.

SUGGESTED RETAIL PRICE: $49.95

Reprinted from InCider Magazine, January, 1987



Power Users' ProDOS

KIX 1.1

Kyan Software Inc., 1850 Union Street #183, San Francisco, CA 94123

ProDOS interface/operating-system shell; 128K IIe, IIc, IIGS

$49.95

Rating: ■■■■

When Apple Computer introduced its first disk-operating system for the old Disk II, it started a tradition that endures to this day: Apple operating systems are weird. Whether you have a Macintosh or an Apple II, Apple doesn't let you interface directly with the operating system. To anyone who has ever used another brand of computer, that's the best evidence of Apple's peculiarity. MS-DOS, CP/M, VAX/VMS—they all let you enter commands at the keyboard to manage and manipulate your disks and files. Apple keeps you at arm's length, with menu interfaces like those of the ProDOS Filer.

KIX, an operating-system enhancement from Kyan Software, changes that. It extends ProDOS by adding disk-based commands to the system. Keeping commands on disk until they're needed is common practice with other operating systems, but foreign to the Apple II, where the entire DOS is kept in memory at all times. This is certainly convenient, but it limits the power of the system.

By contrast, KIX gives ProDOS users access to many features of more powerful operating systems such as MS-DOS and UNIX. If your Apple II system includes a high-capacity disk drive (UniDisk 3.5 or hard disk), you should buy KIX.

Using KIX

If KIX.SYSTEM is the first system file on your boot disk, KIX will load automatically. Otherwise, you can start KIX by calling KIX.SYSTEM from BASIC or from an applications launcher such as Catalyst. You can also install KIX in AppleWorks, where you access it with Solid apple-K. You exit the KIX environment by typing the command QUIT or entering the pathname of another application.

The interpreter that digests KIX commands occupies less than 1K of memory when active. The commands are stored in a directory named /BIN on the KIX disk. KIX also gives you the option of loading as many commands as will fit into the /RAM disk, to greatly speed up execution.

Some KIX commands replicate ProDOS commands. MKDIR (Make Directory), for instance, mimics the ProDOS CREATE command; LS (List) performs the same function as CAT and CATALOG, but with many more options. You can, for instance, list volumes on line or list all files in a directory, including those in subdirectories. As with some other KIX commands, you can use a greater-than symbol (>) to redirect output of the LS command to any slot; LS>1 lists the contents of the current directory to a printer connected to slot 1.

The advantage of KIX comes from the new commands it adds to ProDOS—specifically its file- and disk-copying commands. No longer do you have to resort to Filer or System Utilities to perform basic operating-system tasks. The KIX CP (copy) command, for example, lets you copy a file from anywhere to anywhere. Using the powerful wildcard options built into KIX, you can copy lots of different files with one command. My only complaint about the CP command is that it copies files only; it won't create a directory as part of the copy process—a shortcoming which, Kyan says, the next KIX will fix.

KIX contains a FORMAT command and a volume-copy command (CPV). Unlike some other operating systems (MS-DOS, for example), KIX warns you if you're about to format a large disk. This should help prevent accidental hard-disk reformats. In all, KIX contains more than two dozen commands and utilities that are a welcome sight to anyone who has to manage files on a hard disk.

KIX and AppleWorks

Since KIX is a power user's product and AppleWorks is the definitive power user's program for the Apple II, I spent a lot of time examining the marriage between KIX and AppleWorks. As a pop-up program within AppleWorks, KIX not only offers disk-handling commands but lets advanced users run Pascal and Assembly programs—creating, in effect, their own Pinpoints. It won't work with the real Pinpoint or other add-ons like MacroWorks. KIX does work with popular third-party memory cards like RamWorks and MultiRam; you simply install KIX after modifying AppleWorks to recognize these memory cards.

When you access KIX from AppleWorks, KIX saves your desktop to disk before letting you enter KIX commands. This keeps the commands from clobbering your data. The problem is that saving a desktop is time-consuming, especially on a floppy-based system; saving to a RAM disk is much more practical. Armed with KIX, Applied Engineering's AppleWorks Desktop Expander and ProDrive software (version 5.2B), and an unaltered version of AppleWorks—not an easy thing to find these days—I set out to create the ultimate AppleWorks-KIX installation, one with which I would still have an expanded desktop and with which I could save the desktop to a RAM disk when invoking KIX.

Surprisingly, I was successful. First, I patched AppleWorks using the AE Desktop Expander. Next, I used the AE Partition utility to let ProDrive (the RAM-drive software) use only 256K of my 512K for a RAM disk. I also used Partition to let my modified AppleWorks recognize only 256K of memory. I then installed KIX. Finally, I formatted a new disk and copied ProDOS, BASIC.SYSTEM, the modified ProDrive, and an Applied-specific startup program (supplied with KIX) to the blank disk. I booted my system with this disk, inserted my AppleWorks Startup, and entered /APPLEWORKS/APLWORKS.SYSTEM. I kept my fingers crossed as my thrice-modified AppleWorks loaded into memory.

Everything worked. When I accessed KIX from AppleWorks, my desktop was saved to the RAM disk in a flash, and recovered just as quickly when I had finished. True, I lost half of my desktop space, but with memory so cheap, I can afford to buy more if I need it. AppleWorks performed fine, and didn't confuse the RAM disk with main memory. I was ecstatic to see products from three different suppliers working so well together.

I don't have many complaints. KIX reports disk errors with codes hidden in the *ProDOS Technical Reference Manual*. I found one mistake in the documentation (128 pages, no index) concerning the proper prefix to use with the RAM-disk AppleWorks Startup, but the solution was easy to find.

More seriously, I missed some method of batching commands together in a file and then executing the file, such as moving my data files into a RAM disk before starting an application. In effect, I could use KIX to create a RAM-based desktop for any application. KIX needs a batch command—again, Kyan says, coming in future versions. There are also plans for a 16-bit GS version with even more UNIX functions.

Overall, though, KIX provides the ProDOS control Apple left out. If you have only 5¼-inch floppies, you don't need it, but the bigger your drives, the more important KIX becomes. Serious users shouldn't be without it.■

Robert M. Ryan
Sharon, NH

InCider's Ratings

Excellent—remarkable, a must buy	■■■■■
Very good—impressive and recommended	■■■■
Good—average, solid performance	■■■
Fair—flawed but adequate	■■
Poor—unacceptable or unusable	■

January 1987

UPDATE ... KYAN

Vol. 2, No. 5 *The bimonthly journal of Kyan Pascal programming* July/August 1987

UPDATE

Latest Software Versions

Title	Version
Kyan Pascal for the Apple	2.02A
Kyan Pascal PLUS Disk 1:	2.02A
Disk 2:	1.02
System Utilities Toolkit, MouseText Toolkit, Advanced Graphics Toolkit, TurtleGraphics Toolkit	1.00
Code Optimizer Toolkit	1.01
Font Utilities Toolkit	1.00A
KIX	1.01

All of Kyan's software packages for the Apple remain unchanged since the last issue of UPDATE...KYAN. To update, send your disk(s) plus $10 for each software title you wish to update. (Note that although Kyan Pascal PLUS has two disks, it still only costs $10 to update.)

Real Programmers Don't...

compiled and uploaded to Usenet by Sean Philip Engelson, Carnegie-Mellon University Computer Science Department

reprinted fromThe ACORN Kernel newsletter

• Real programmers don't write specs—users should consider themselves lucky to get any programs at all and take what they get.

• Real programmers don't comment their code. If it was hard to write, it should be hard to understand.

• Real programmers don't eat quiche. In fact, real programmers don't know how to *spell* quiche. They eat Twinkies and Szechwan food.

• Real programmers don't write in COBOL. COBOL is for wimpy applications programmers.

• Real programmers' programs never work right the first time. But if you throw them on the machine they can be patched into working in "only a few" 30-hour debugging sessions.

• Real programmers don't write in FORTRAN. FORTRAN is for pipe stress freaks and crystallography weenies.

• Real programmers never work nine to five. If any real programmers are around at 9a.m., it's because they were up all night.

'Real programmers don't eat quiche. In fact, they don't know how to spell quiche.'

• Real programmers don't write in BASIC. Actually, no programmers write in BASIC, after the age of twelve.

• Real programmers don't write in PL/1. PL/1 is for programmers who can't decide to write in COBOL or FORTRAN.

• Real programmers don't play tennis or any other sport that requires you to change clothes. Mountain climbing is okay, and real programmers wear their climbing boots to work in case a mountain should suddenly spring up in the middle of the machine room.

• Real programmers don't document. Documentation is for simps who can't read the

listings or the object deck.

• Real programmers don't write in PASCAL or BLISS or ADA, or any of those pinko computer science languages. Strong typing is for people with weak memories. *(See following article for a scathing rebuttal. —Ed.)*

• Real programmers never make up schedules. Only planners make up schedules. Only managers read them.

• Real programmers never deliver programs on schedule. Either the program is "done" in two days or it is never finished. In any case, it is never delivered when it was scheduled.

• Real programmers never eat at restaurants. If the vending machine sells it, they eat it; if it doesn't, they don't. Recently real programmers discovered that popcorn was being sold in vending machines. Common coders discovered that it could be popped in the microwave oven in the vending machine room but real programmers use the heat escaping from the top of the CPU.

• Real programmers never deliver programs on Wednesdays.

• Real programmers never deliver programs on the first day of any month.

• Real programmers know that good human factors design requires only the application of common sense. Besides, no one cares about users. The program's written for aesthetic beauty.

• Real programmers know every nuance of every instruction and they use them all in every program.

• Real programmers do not clear registers twice before using them. In fact, if you anoy a real programmer, he/she won't clear the registers at all. And that goes for your memory too!

• Real programmers do not wonder where the bits went following a shift operation. They do not care.

• Real programmers are not in it for the money. Most of them are secret millionaires.

More Changes

Yes, we did it again. While you weren't looking we sneaked this spiffy new newsletter format in behind your back.

You can't say we didn't warn you, though. We finally got around to attempting to work with PageMaker and it's not as hard as it looked. Only trouble was that they released a brand-spanking-new v2.0 of PageMaker and the newsletter had to be totally redone after half of it was already completed with the old version, so we apologize for any delay.

Yes, Virginia, there really *is* such a thing as 'desktop publishing!'

Real Programmers Do...

• Real programmers do consume mass quantities of Jolt cola. ("All the sugar and twice the caffeine of regular colas!") This is because valuable programming time may be spent in jail for possession of methamphetamines, so they settle for the next best thing.

• Real programmers do listen to Led Zeppelin, Aerosmith, Van Halen, and Megadeth (and generally any other loud, fast music) at ridiculously high decibel levels in order to keep them awake through the long programming sessions.

• Real programmers do know every single ASCII character's corresponding numerical value, not only in decimal, but also in hex, octal, and binary.

• Real programmers do wear only jeans and T-shirts. A pair of slacks is permissible, a tie is pushing it, and a suit is sacrilegious. They avoid meeting with the tie- and suit-wearers. Except on payday.

• Real programmers do hate mice. (Not

the kind that scurry along the floor, but the kind that are supposed to roll along a desktop, which is impossible because a real programmer's desktop is covered with tech notes, quick reference cards, day-old pizza, and Jolt cola.)

• Real programmers do know that pointers have to do with addresses, not with said hated mice.

• Real programmers do use quick reference cards instead of online help screens. Quick reference cards may take up desk space, but help screens take up memory space, a far worse crime. (Besides, as real programmers say, "only sissies need help.")

• Real programmers do love CONTROL, ALT, and function keys and praise macros.

• Real programmers do use command line programming environments with at least twenty commands and at least two switches or options for each command. Menus and windows are for sissies.

• Real programmers do write long, nasty letters to software publishers detailing bugs, demanding fixes and cutting down the programmer of the software for writing wimpy code. They do not call technical support lines with questions, in fear of appearing as though there might be something about programming they do not know.

• Real programmers do spend their lunch hours at school programming instead of eating.

• Real programmers do spend their lunch hours at work programming instead of eating.

• Real programmers do often appear anorexic.

• Real programmers do always use subsets of longer words. For example, 'tech,' 'specs,' and 'docs' for 'technical,' 'specifications,' and 'documentation,' respectively.

• Real programmers do invent terms for menus, windows, and help screens so important to the users. For example, 'Sissy Screen' and 'Wimp Window.'

• Real programmers do always type "Pascal" properly capitalized and in lower case—never "PASCAL" in upper case like the common-coders. *(See previous article. --Ed.)*

• Real programmers do spend what little time they have looking for new hardware and software.

• Real programmers do spend what little money they have purchasing new hardware and software.

• Real programmers do often have resentful spouses/lovers. (Or none at all.)

• Real programmers do get into the business mainly so that they can say things like 'post-mortem symbolic debugger.'

PASCAL PROGRAMMING

Using Characters as Bytes

A Good Reason to Use Them

Because your programs seem to take up more and more memory space, it might seem wasteful to use Integers when poking values into and peeking values from memory because an Integer takes up two bytes of memory. (And actually, it *is* wasteful!)

For example, let's assume that we want to poke the number 1 into memory location 752. Here is how this is done:

```
VAR
   memloc : ^integer;
(*pointer to integer*)
    value : integer;
      .
      .
      .
value := 1;
(*assign the value*)
memloc := Pointer(752);
(*specify the address*)
memloc^ := value;
```

```
(*perform the poke*)
```

Because memloc is a pointer to an Integer, it requires two-bytes for the pointer address and another two bytes for the Integer which it points to. value, although we never will have it exceed 255 (the maximum number one byte can hold), hogs up two bytes of memory, too. As you can see, a few bytes are being wasted.

Kyan Pascal has no Byte data type which we need for value in this case, but there is something almost as good—Char. "What!?" you say, "Poke a character into an address? Yeah, right" Strange as it may seem, it can be done. Here's how:

```
VAR
   memloc : ^char;
   value : char;
      .
      .
      .
value := Chr(1);
(*assign the value*)
memloc := Pointer(752);
(*specify the address*)
memloc ^ := value;
(*perform the poke*)
```

We can use characters because the computer represents a character with a number; it has no other way of doing things. This gives us the ability to perform a poke with a character because in reality, it is just a number. By making memloc a pointer to a character, a byte is spared; by making value a character, another byte is spared. Notice how the predefined Chr function is used to assign value the ASCII character corresponding to 1—you needn't know or care about the ASCII character, only the number.

We can perform a peek in much the same way with:

```
memloc := Pointer(752);
(*specify the address*)
value := memloc ^;
(*perform the peek*)
WriteLn('Peeking into 752 reveals the
number ',Ord(value));
```

Another Good Reason

Depending on the type of programming you do, the space saved by this 'char poke' technique may be negligible to you; if so, character poking has another benefit which offers potentially greater utility.

Normally, when you perform a poke in Pascal, either with the Poke routine in the Assembler section of the Kyan Pascal User's Manual or with the Pointer-to-Integer technique, there exists the potential to adversely affect your program. Because an Integer takes two bytes of storage, it writes *both* of these bytes into memory. For example, this Pointer-to-Integer technique:

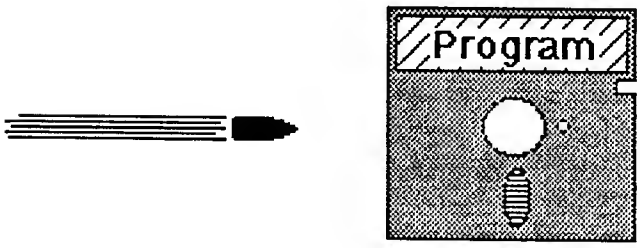
```
value := 1
memloc := Pointer(752);
memloc^ := value;
```

puts the least significant byte (LSB) of the Integer into memory location 752, as we would expect. Unfortunately, Mr. Murphy, that little guy who makes everything go wrong, also puts the most significant byte (MSB) of the Integer, 0 in this case, into the *following* memory location, 753 in this case.

Now it just may be the case that with your particular operating system, poking 1 into memory location 752 causes your computer to write the meaning of life and the secret to eternal happiness to your screen but poking 0 into address 753 causes your computer to launch several multiple-warhead thermonuclear ICBMs. Now if this were true, wouldn't it be a real bummer to find out the meaning of life and the secret to eternal happiness and then be Chernobylized fifteen minutes later?

(The moral to this story is: Use the Char-Poke Technique When Unsure of What the Following Address Controls —or— Never Use the Pointer-to-Integer Technique if You Work for the Pentagon.)

(Character-oriented Peek and Poke routines appear at the end of the newsletter in the LISTINGS section. —Ed.)



Bulletproofing Programs

It used to be that in the old days, users of programs were pretty knowledgeable of their computers and when an error or crash occurred they would not complain much. (In those days software was hard to come by and users didn't want to upset the programmers.) These days however, program users are getting less and less knowledgeable of their machines and more and more upset about crashes, much to the disappointment of programmers.

When writing their programs, programmers must keep in mind that end users may know very little about computers and what makes the programs crash. For example, a user may insert a data disk into the disk drive *sideways* (stifle those giggles; the author has seen it happen), which will usually result in some sort of fatal I/O error when your program attempts to read from or write to the disk. A more common cause of runtime errors is entering letters when the program expects numbers, as even the best typist may inadvertently tap the semicolon key just before the carriage return. There are scores of things that can go wrong with even the simplest of programs, so the programmer must be prepared for them.

To make a program truly impossible to crash (also known as 'bulletproof,' 'waterproof', and 'idiot-proof'), the programmer must keep in mind Murphy's law because it applies so well to programs. Whatever *can* go wrong *will* go wrong. This means that the programmer must keep a wary eye open to watch any place in the program which provides an opportunity for an error. A great percentage of the errors happen during input/output for a number of reasons, such as:

- Input sometimes requires the user to type in data from the keyboard, which is an inaccurate means of data entry
- I/O devices, unlike the computer itself, are often mechanical and have moving parts which are more prone to failure
- I/O devices are subject to the user's correct operation (for example, online/offline buttons on printers and doors or latches on disk drives)
- A file may not be found on disk when reading or the disk may be too full to write

Therefore, the would-be bulletproof program must take nothing for granted and must always assume that an error can and will occur during I/O.

Bulletproofing a program is accomplished by trapping the possible errors. For example, if a program requires the user to input a number

```
Write('Enter rate of speed: ');
ReadLn(speed);
(* assume speed : integer *)
```

the user may type in "55mph" not knowing that the "mph" characters are unnecessary and will cause the program to crash with an input error. Or the user, being a stickler for accuracy, may enter "55.42" which will also cause a crash. (See what happens when you exceed the legal speed limit?)

To trap an input error in this situation, a few things should be done. First, the user should

know, from the program's documentation, to enter an integer and nothing else. (In this case, it would also be helpful to ask for miles per hour instead of kilometers or the program may have to face an input of "88" when a metric-oriented person steps up to the keyboard.) The program should read the input in as a string, because practically anything can be entered from the keyboard in a string, with the possible exception of the Reset key. The program should then convert the string to the number needed. The System Utilities Toolkit contains several number-to-string and string-to-number conversion routines which will do the job just fine. So, with these things in mind, the code might look like this:

```
Write('Enter rate of speed: ');
ReadLn(in_string);
speed := StrToInt(in_string);
```

That is a painless, efficient way of trapping input errors from the user. The errors such as the one demonstrated are called type-conflicts or type-clashes and because Pascal is a strongly-typed language, the compiler will flag and report a conflict such as:

```
character := real_number + 1;
```

and a similar error will crash the program during runtime. The other kinds of errors are more difficult to trap, as we will soon see.

'If a program crashes, it is dead... not even Oral Roberts can save it.'

THE FURTHER ADVENTURES OF JOE PROGRAMMER

Let's suppose that a gentleman named Joe Yoozer is using *TurboCalcWriteCommGraph*, a $99.95 do-everything program written in Kyan Pascal by Joe Programmer and published by JoeSoft, Inc. *TurboCalcWriteCommGraph*, being a typical Joe Programmer program, makes no provision for trapping errors of any kind. Let's also pretend that eight hours into entering kilobytes of data into the Calc module Joe Yoozer, being a typical software user, *finally* decides it's about time to save his data to disk. However, Mr. Yoozer inserts an unformatted disk into the drive, the program crashes while attempting to write to the disk, and Joe Yoozer becomes very upset over having just lost eight hours of work.

After a call from an irate Joe Yoozer, Joanne Texupport of JoeSoft reports the problem to Joe Programmer and he decides to add some error trapping and issue an update to JoeSoft's best-selling program.

Joe Programmer soon realizes that trapping errors with I/O devices, unlike simple keyboard data entry errors, is very difficult. As the programmer, he has no control over whether or not the user properly inserts a formatted diskette with enough disk space for the file or whether or not the file specified is on the disk, and so on. A Reset or Rewrite statement can be very unforgiving when it comes to I/O errors. For example, if the statement

```
Reset(Fyle,'/Vol/Myfile');
```

were executed and Myfile were not on the Vol volume, the program would crash with an I/O error message. That is what the ISO rules say must happen—no ifs, ands, or buts. The program is dead; not even Oral Roberts can save it.

So Joe Programmer, it may seem, is stuck. The only practical, truly-effective way to trap disk errors is to go into the assembler and program with the MLI, the ProDOS Machine Language Interface. After an input or output operation, the MLI returns an error status byte which can be checked and acted upon by the programmer. Joe will have to write a whole new set of disk I/O procedures and functions using assembly language. Because the Pascal runtime library already takes up a good deal of memory for I/O routines, this double use of memory is somewhat wasteful.

However, there is a light at the end of the

tunnel, and it is the Code Optimizer Toolkit. Besides the speedy and memory efficient macros it gives your program, it also contains the source code to the runtime library. A good programmer can rewrite the normal input and output routines of the runtime library so that I/O errors can be detected and acted upon as he deems necessary so that crashes do not happen. However, rewriting these assembly language I/O routines is not for the novice programmer; only an intermediate to advanced programmer, such as Joe Programmer, should attempt these modifications.

So, using the Code Optimizer Toolkit, Joe Programmer tweaks his program so that it won't crash upon experiencing a runtime I/O error. With this and the other added feature of greater speed given by the Optimzer, JoeSoft releases a software update: *TurboCalcWriteCommGraph* v1.1. (Have your credit card ready; operators are standing by to take your order!)

Bulletproofing Tips

- Inform the user

The user must know what types of data to enter and what not to

- Expect the unexpected

If there is a possibility that an I/O error will occur, watch for it

- Accept only good input

Read in a string and convert it to a number later

LISTINGS

```
PROCEDURE CharPoke(charpoke_addr : integer;
charpoke_val : integer);
(* by Erik Warren, 17 July 1987 for
UPDATE...KYAN Vol.2, #5 *)
(* not the most efficient way to poke but
it demonstrates how to use chars *)
(* call it like this:
CharPoke(myaddress,myvalue);  *)
  VAR
     charpoke_pointer : ^char;

BEGIN
  charpoke_ptr := Pointer(charpoke_addr);
  charpoke_ptr^ := Chr(charpoke_val)
END; (* of CharPoke procedure *)
```

```
FUNCTION CharPeek(charpeek_addr : integer)
: integer;
(* by Erik Warren, 17 July 1987 for
UPDATE...KYAN Vol.2, #5 *)
(* not the most efficient way to peek but
it demonstrates how to use chars *)
(* call it like this:  myvalue :=
CharPeek(myaddress);  *)
  VAR
     charpeek_ptr : ^char;

BEGIN
  charpeek_ptr:= Pointer(charpeek_addr);
  CharPeek := Ord(charpeek_ptr^)
END; (* of CharPeek function *)
```

UPDATE...KYAN is designed and edited by Erik Warren and published bimonthly by Kyan Software, Inc., 1850 Union Street, Suite 183, San Francisco, California 94123

Tech Notes and Errata

ASSEMBLY LANGUAGE PROGRAMMING

Changes to Routines Used in Kyan Pascal, Version 1.-

Assembly language routines written for version 1.- of Kyan Pascal must be converted to run under the new compiler environment. The following changes are required.

1. The local variable stack now begins allocating storage at SP+5 instead of SP+3.

2. The predefined labels T, SP, and LOCAL are now redefined as _T, _SP, and _LOCAL. Note the underscore character preceding the label.

3. The new compiler uses the underscore-first convention to distinguish compiler/assembler system labels from user labels. **DO NOT USE THE UNDERSCORE IN YOUR OWN LABELS.**

4. Routines which use RAM bank 2 of Motherboard ROM will destroy the KIX shell. The references must be removed from your programs.

5. Routines which use memory $300 through $340 will disable KIX. All "safe" storage must be done in user-defined data areas in the host Pascal program.

6. If you own an Apple II or II+, you can simulate the underscore character by pressing **<Control> - Shift - P**.

Functions Written Entirely in Assembly Language

Conformance to the ISO standard dictates that the compiler return an error when it encounters a FUNCTION written entirely in assembly language. This occurs because ISO requires any FUNCTION identifier to be explicitly assigned a return value (something which obviously does not happen in pure assembly language routines). For example:

```
FUNCTION DEMO:BOOLEAN;
BEGIN
#A
        LDY #5              ; STACK ADDRESS OF FUNCTION VALUE "DEMO"
        LDA #1              ; TRUE BOOLEAN VALUE
        STA (_SP),Y         ; MAKE 6502 ASSIGNMENT
#
END;
```

This function will generate the error message: *FUNCTION RESULT UNDEFINED.*

The solution to this problem is to declare a local variable which matches the type of the function being written. For ease of use, be sure the local variable is the last one declared so that it is always at the top of the local stack space (i.e., starting at offset 5). Instead of storing the function result directly into the stack space allocated for it, put the result value into the last local declared (mentioned above). Then, the last line of the function code before the END; is an assignment of the function identifier to the local variable at the top-of-stack. This method makes machine code clearer to other programmers and makes it easier to debug a function.

For example:

```
FUNCTION GOODEXAMPLE : BOOLEAN;
VAR RESULT : BOOLEAN;
BEGIN
#
;
; PERFORM CALCULATIONS HERE
;
        LDY #5                  ; ALWAYS STACK POSITION OF LAST DECLARED
        STA (_SP),Y               LOCAL VARIABLE
#
        GOODEXAMPLE := RESULT
END;
```

PREDEFINED FUNCTIONS AND PROCEDURES

The ASSIGN statement has been replaced by ADDRESS and POINTER as predefined functions. This modification to Kyan Pascal gives the programmer more power and flexibility. The syntax and description of these functions follows.

ADDRESS

Purpose: This function returns the memory location of the first storage location of the identifier passed as an integer value.

Syntax: FUNCTION ADDRESS(identifier):INTEGER;

Example:

```
PROGRAM EXAMPLE
VAR X : INTEGER;
BEGIN
        WRITELN ('The address of variable X is ', ADDRESS(X))
END.
```

POINTER

Purpose: Assign a pointer an integer value. Note that the integer passed does not have to be constant (i.e., it can be a formula).

Syntax: FUNCTION POINTER(X:Integer) :INTEGER;

Example:

```
PROGRAM EXAMPLE2;
VAR     X : INTEGER;
        Z : ^INTEGER;
BEGIN
        X := 100;          (* NOTE THE MSB OF X IS 0 *)
        Z := POINTER(ADDRESS(X));
        WRITELN(Z^)
END.
```

This program will return the output "100".

HiRES GRAPHICS and RUNTIME MEMORY

Since this manual was printed, the procedure for relocating memory for HiRES graphics has been changed. Instead of using the directive "ORG $4000" to relocate the program in memory (see page III-15), you should use the directive "_UsesHires". The impact of this new directive is illustrated in the following modification of the Runtime Memory Map.

_SystemFiles are loaded at $2000 and relocated depending on the setting of "_UsesHires" (on or off). The "_UsesHires" causes the BIN image of the object program to be loaded at location $4000 and the heap to occupy from $800 to $2000, thus leaving $2000 - $3FFF clear for the HiRES routines.

Runtime Memory Map with " _UsesHires" **ACTIVE**

0	-	$ 7FF	Apple system overhead
$ 800	-	$1FFF	System variable space
$2000	-	$3FFF	Hi-Resolution Graphics (page 1)
$4000	-	$8FFF	Program
$9000	-	$BEFF	_LIB (Kyan Pascal Runtime Library)
$BF00	-	$BFFF	ProDOS primary access page
$C000	-	$CFFF	Soft switches/peripheral ROM space
$D000	-	$F7FF	Appiesoft BASiC
$F800	-	$FFFF	System Monitor

Runtime Memory Map with " UsesHires" **INACTIVE**

0	-	$ 7FF	Apple system overhead
$ 800	-	_LoMem	Program
_LoMem	-	$9000	System variable space
$9000	-	$BEFF	_LIB
$BF00	-	$BFFF	ProDOS primary access page
$C000	-	$CFFF	Soft switches/peripheral ROM space
$D000	-	$F7FF	Applesoft BASIC
$F800	-	$FFFF	System Monitor

Announcing ...
The Font Utilities Toolkit

Kyan Software is proud to introduce the **Font Utilities Toolkit**, the latest in a line of toolkits which make programming with Kyan Pascal faster and easier.

The **Font Utilities Toolkit** is a family of routines which let you add text to the hi-res screen. It lets you jazz up your application programs with custom graphics, large fonts, and animation! With the Toolkit, you can create Kyan Pascal application programs with hi-res displays featuring:

CUSTOM CHARACTER FONTS
which are
LARGE or **SMALL**, **FORMAL** or *Hang-Loose*

With the **Font Utilities Toolkit**, you can create and save your own custom character sets. Build a custom font library and use them in other programs!

AND, you can move characters about the screen. Animate your program displays!

Using the **Font Utilities Toolkit** is simple. Instead of cryptic hex addresses and bitmaps, you can use clear English-like names and syntax such as:

```
PutChar('A', At[Column,Row]);
```

And, if you are a developer interested in fast, compact code, the **Font Utilities Toolkit** lets you make the font routine calls in assembly language.

The **Font Utilities Toolkit** was developed and licensed to Kyan Software by Kevin Neelands, Senior Programmer, Acumen Development.

Hardware Requirements:	**Any Apple // with at least 64K**
Software Requirements:	**Kyan Pascal, Version 2.0**
Retail Price:	**$29.95**

ORDER FORM

YES! Please send me ____ copies of the **Font Utilities Toolkit** for my Apple //. I am enclosing $29.95 for each copy plus $4.50 for shipping and handling ($15 outside North America) for a total of $:_____

Payment Method (please mark one): ___ check/money order ___ Visa ___ MasterCard

Card Number:_____ Exp. Date:_____

Name:_____ Phone:_____

Address:_____

City:_____ State/Province:_____

ZIP/Postal Code:_____ Country:_____

SEND YOUR ORDER TO: **Kyan Software Inc., 1850 Union Street #183, San Francisco, CA 94123**

OR CALL: 415-626-2080

APPLE // SOFTWARE CATALOG

kyan

KYAN SOFTWARE INC.
SAN FRANCISCO, CALIFORNIA

kyan

kyan software
1850 union street #183
san francisco, california 94123

First Class Mail

Kyan Software Inc.
1850 Union Street, #183
San Francisco, CA 94123
(415) 626-2080
Telex: 989113 KYANSFO

Catalog No. 2
Fall, 1986

MCI: 298-0892
CompuServe: 73225,450

Dear Friend:

This catalog describes Kyan Software's complete line of programming tools and utilities for the Apple // family of computers. These Kyan products run under ProDOS and provide you with the tools necessary to develop the latest state-of-the-art software. These products are now available from Kyan Software or your local software retail store.

There are many good reasons for you to buy Kyan Software products. Following are just a few:

Kyan software is not copy-protected. You can make backup copies and copy the software into RAMdisk or onto hard drives without limitation.

Kyan software is royalty free. You can use the Pascal Runtime Library and routines in the programming Toolkits without paying royalties. (Note: Copyright notices must be observed and some modules must be licensed directly from Apple Computer Inc.).

Kyan provides support. Our technical support staff is one of the best in the business. We are ready to help you in selecting the right programming tools and working through programming problems. Coupled with Kyan's newsletter, **Update ... Kyan**, Kyan keeps you abreast of the latest programming tips and software developments.

Kyan offers periodic low cost upgrades. When product revisions are released, existing owners are offered upgrades at little or no charge. When new products are introduced, existing owners find out about them first, and are frequently offered a substantial discount from the retail price.

Source code compatibility and portability. Since Kyan Pascal is ISO standard Pascal, the software you write on your Apple is portable to most mainframe computers. Also, since Kyan has similar compilers for the Atari and Commodore family of 8-bit computers, the source code for your Apple // programs can be recompiled to run on these other popular computers.

Finally, Kyan Software guarantees your satisfaction. All software ordered directly from Kyan comes with a 30 day money-back guarantee. If you are not completely satisfied, return the product for a full refund of the purchase price.

Join the Kyan Software family today!

Kyan Products for the Apple II Family

Kyan Pascal
(Version 2.0)

Kyan Pascal is the perfect package for learning Pascal and developing Pascal programs. It is a full implementation of ISO Pascal and will run on any Apple II with 64K of memory and a single disk drive. It is widely used by both students and advanced programmers because of its economical price, user-friendly environment, and extensive list of features.

Kyan Pascal is an extremely powerful programming tool, yet it keeps the beginner in mind. It features command menus, HELP screens and several libraries of error messages. It comes with a comprehensive, 300 page manual which includes a complete Pascal tutorial. It is perfect for students who are taking a Pascal programming course and want do their assignments at home.

But, **Kyan Pascal** is more than just a learning tool. It's also a powerful software development system. It provides experienced programmers with features and capabilities not found in other Pascal products. And, when teamed with Kyan's new programming toolkits, **Kyan Pascal** gives you the ability to add MouseText, MouseGraphics, 3-dimensional graphics, and other advanced features to the programs you write.

Other **Kyan Pascal** features include:

- o ProDOS Operating System included on the disk.
- o 6502 machine code compiler and assembler, which generate code that runs more than 30 times faster than BASIC.
- o Full Screen Text Editor with 40 and 80 column modes.
- o Pascal extensions like include, chain, string handling, random files, random numbers, and high resolution graphics.
- o Built-in assembler which allows you to add in-line or included assembly source code.
- o Non-copy protected disks that allow you to make backup copies and load files into RAM, 3.5" Unidisk, or hard disk drive.
- o Royalty-free license to use the Pascal Runtime Library with software you develop.

Try Kyan Pascal. Find out why it is quickly becoming the standard for Pascal programming on the Apple II.

Hardware Requirements: Any Apple II with 64K of memory
Suggested Retail Price: $69.95

KIX™

"Adds the power of MS-DOS and UNIX to ProDOS"

If you use your computer a lot, you know how limited ProDOS is as an operating system. With **KIX** you can unchain ProDOS and give it the power of MS-DOS and UNIX. **KIX** gives you capabilities that have been available to IBM and other PC users for years.

KIX is designed specifically for power users of the Apple II. It is a powerful ProDOS shell which saves you time and wasted motion. With simple command lines, you create or delete directories, manipulate files, search documents for key words, lock sensitive files, and much, much more. **KIX** has hundreds of command options. And, they can be called from a system prompt or from inside AppleWorks!

If you've never worked with a real PC operating system, it may be difficult to understand how it can simplify your life and increase your productivity. If you have, you've probably grumbled to yourself many times about the lack of power in ProDOS. Well, grumble no more!

KIX gives you the power to:

Control Directories: (print, create, delete or change directories);

Manipulate Files: (copy, move, delete, rename, print or change protection status of files);

Work with Volumes: (copy, delete, or rename volumes; format disks);

Compare Files and Volumes: (compare and list the differences between multiple files and/or volumes);

Search Files and Directories: (search directories for files and/or search files for character strings); and,

Make Hardware/Firmware Calls: (set system time/date and 40/80 column video outputs; and dump screen contents to a printer).

In addition, KIX contains full support for wildcards and redirection of output to printers and other peripheral devices.

Hardware Requirements: Any Apple // with 64K and two disk drives; RAMcard recommended.
Suggested Retail Price: $49.95

Kyan Pascal PLUS

Kyan Pascal (version 2.0) is perfect for learning Pascal and developing Pascal programs. It is a full implementation of ISO Pascal and runs under ProDOS on any Apple II (with 64K of memory and a single disk drive). It is widely used by students and advanced programmers alike because of its economical price, user-friendly environment, and extensive list of features.

Kyan Pascal PLUS is a powerful programming tool, yet it keeps the beginner in mind. It comes with a comprehensive (480 page) manual containing a Pascal tutorial, Quick Reference Guide, and complete instructions.

But, **Kyan Pascal PLUS** is more than just a learning tool. It's also a sophisticated software development system. **Kyan Pascal PLUS** includes **KIX™**, Kyan's powerful new UNIX-like environment for the Apple II. **KIX** gives programmers more than 25 operating system utilities which greatly enhance programming speed and productivity. And, when teamed with Kyan's new Programming Toolkits, **Kyan Pascal PLUS** gives you the ability to add MouseText, MouseGraphics, 3-D graphics, and other features to your programs.

Kyan Pascal PLUS also features:

- o ProDOS Operating System included on the disk.
- o 6502 machine code compiler and macro-assembler, which generate code that runs more than 30 times faster than BASIC.
- o Full Screen Text Editor with 40 and 80 column modes.
- o Pascal extensions like include, chain, string handling, random files, random numbers, gotoxy, and high resolution graphics.
- o Built-in assembler which allows you to add in-line or included assembly source code
- o Non-copy protected disks that allow you to make backup copies and load files into RAM, 3.5" Unidisk, or hard disk drive.
- o Royalty-free license to use the Pascal Runtime Library.

When you try **Kyan Pascal PLUS**, you will quickly discover why it is becoming the standard for Pascal programming on the Apple II.

Hardware Requirements: Any Apple II with 64K of memory
Suggested Retail Price: $99.95

***Note:** Kyan Pascal PLUS does not include the KIX-AppleWorks feature. If you want this feature, please order Kyan Pascal (Version 2.0) and KIX separately.*

Toolkit I
System Utilities

The **System Utilities Toolkit** contains Pascal and Assembly language routines which are designed to be used in your Pascal application programs. The Toolkit contains 74 routines plus sample programs and complete instructions for using each routine in your own programs. The Toolkit routines are organized into four libraries.

I. ProDOS Utility Library

This library contains routines which let you access the functionality of various ProDOS functions and procedures from within your Pascal program. The library contains 27 routines including procedures for: copying, renaming, deleting, locking/unlocking, and appending files; creating, removing, and searching directories; setting a system clock; and much, much more.

II. Device Driver Library

This library contains 18 functions and procedures which allow you to link your application programs and external devices. The library includes routines for mouse, joystick, and trackball applications.

III. Screen Management Library

This library contains 20 routines which you can use in your Pascal programs to control screen functions. The library includes routines for: GOTOXY and other cursor control functions; scrolling, inverse and clear screen actions; and, on/off routines for 80 column firmware. It also includes a routine which identifies the hardware configuration in which your program is being booted (e.g., Apple IIe with 80 columns vs. Apple II+ with 40).

IV. Other System Utilities

This library includes random number routines plus: conversion routines which allow you to convert strings to integers or real numbers and vice versa; sort and merge routines which allow you to sort records and files alphabetically or numerically; and a line parsing routine which enables you to interpret command line inputs to your Pascal programs.

Hardware Requirements: Any Apple // with 64K
Software Requirements: Kyan Pascal, Version 2.0
Suggested Retail Price: $49.95

Toolkit II
MouseText

The **MouseText Toolkit** is a family of routines which provides the experienced programmer with an easy means of adding the Macintosh "look and feel" to programs written for the Apple II. With **Kyan Pascal** and the **MouseText Toolkit**, you can write programs for the Apple II which contain windows with text displays, menu bars, pull-down menus, and mouse-controlled events.

Many of the applications being developed today for the Apple II have this "look and feel." With the **MouseText Toolkit**, you'll have the same tools used by these professional software developers.

The **MouseText Toolkit** consists of a Runtime Module, utility files, and more than 50 MouseText routines. It also contains sample programs which show how to use these routines in your Pascal programs.

To use the **MouseText Toolkit**, just declare the MouseText routines in your Pascal program. Then, as you develop your application, call these routines to create windows and menus, track cursor movements, and read mouse events (e.g., mouse "clicks"). When your program is complete, it will have the Macintosh "look and feel".

A mouse is recommended when using the Toolkit. However, it is not essential. The **MouseText Toolkit** provides routines which allow you to simulate the actions of a mouse using cursor keys.

The **MouseText Toolkit** includes:

- o Demo Programs,
- o Programming Utilities,
- o Cursor Routines,
- o Event-Handling Routines,
- o Menu Routines,
- o Window Routines,
- o Control Region Routines, and
- o MouseText Runtime Module.

Hardware Requirements: //c or //e with enhanced ROM; (mouse recommended)
Software Requirements: Kyan Pascal, Version 2.0
Suggested Retail Price: $49.95

Toolkit III
Advanced Graphics

The **Advanced Graphics Toolkit** is a family of routines which provide the experienced programmer with an easy means of adding sophisticated graphics to Apple // programs. With **Kyan Pascal** and the **Advanced Graphics Toolkit**, you can create HiRes and double HiRes displays (both color and monochrome), 3 dimensional graphics, perspective drawings, and much more.

The **Advanced Graphics Toolkit** supports the Apple // double hi-res graphics display on the Apple IIc and IIe (128K version). With a color monitor, you can display twelve colors plus white, black, and gray.

The **Advanced Graphics Toolkit** consists of two modules -- graphics primitives and advanced (3-dimensional) graphics.

The graphics primitives provide the programmer with a set of basic procedures for creating 2 dimensional graphics displays. The graphic primitives are grouped into the following categories.

Initialization: Commands which set-up the operating environment;
GrafPort: Commands to set-up, activate, or modify GrafPorts
Drawing: Commands to create graphic displays; and,
Text: Commands to place text in graphic images.

The advanced graphics provide the Pascal programmer with tools for 3-dimensional images. The procedures support:

Geometry and Line Generation: Location and display of simple points and line segments;
Polygons (many-sided figures): Display of solid patterns and objects using coloring and shading;
Transformations: Uniform alteration of images through scaling (changing the image size), translation (moving the image), and rotation (rotating the image about a fixed point);
Windowing and Clipping: Selection and enlargement of portions of a drawing, and clipping (deletion) of undesired portions;
3 Dimensional Images: Extension of graphics to 3 dimensions; and,
Curves: Generation of curves using interpolation and B-spline methods.

Hardware Requirements: Apple //c or //e with 128K
Software Requirements: Kyan Pascal (version 2.0)
Suggested Retail Price: $49.95

Toolkit IV
TurtleGraphics

The **TurtleGraphics Toolkit** contains Pascal and Assembly language routines which are designed to be used in your Pascal application programs. The Toolkit contains 24 routines plus sample programs and complete instructions for using the routines in your own programs. The Toolkit routines are organized into three libraries:

I. TurtleGraphics Library

This library contains 17 routines which allow you to use TurtleGraphics in your programs. The Library routines allow you to initialize the turtle, move it around the screen, draw lines in different colors, and more. The routines include:

o InitTurtle	o PenColor	o GrafMode	o TextMode
o Turn	o TurnTo	o Move	o MoveTo
o TurtleX	o TurtleY	o TurtleAng	o ViewPort
o FullPort	o FillPort	o SaveHires	o LoadHires
o FillArea			

II. Sound Effects Library

This library contains four procedures used to generate sound effects in an application program. The routines include:

o Beep	Sounds a beep from the Apple speaker
o Note	Sounds a tone with a specified pitch and duration
o Click	Generates a click from the speaker
o Phaser	Creates a phaser sound effect

III. Chart Routines

This library contains 3 procedures which allow you to graphically display data. The routines include:

o BarChart	Draws proportioned bar graph of data
o PieChart	Generates a pie chart
o PlotXY	Plots an X vs. Y graph, point by point

Hardware Requirements: Any Apple // with 64K
Software Requirements: Kyan Pascal, Version 2.0
Suggested Retail Price: $29.95

Toolkit V
Available 10/15/86
MouseGraphics

The **MouseGraphics Toolkit** is the graphical analog of the **MouseText Toolkit.** Its intent is to bring the Macintosh style of user interface to the Apple II. Many of the applications being developed today for the Apple II have this "look and feel." With the **MouseGraphics Toolkit,** you have access to the same tools used by professional software developers to write these programs.

The **MouseGraphics Toolkit** supports the double hi-res graphics display on the Apple IIc and IIe (128K version). With a color monitor, you can display twelve colors plus white, black, and gray. The procedures in this Toolkit support pull-down menus, windows, cursors, and event-handling.

The **MouseGraphics Toolkit** consists of two modules. The first module consists of the graphics primitives. These primitives provide the programmer with a set of fundamental operations for creating graphics displays. The graphics primitive commands are grouped into the following categories.

Initialization: Commands which set-up the operating environment;
GrafPort: Commands to set-up, activate, or modify the GrafPorts;
Drawing: Commands to create graphic objects in images; and,
Text: Commands to place text in graphic images.

The second module in the **MouseGraphics Toolkit** consists of a complete set of MouseGraphics commands. These are grouped into six categories.

Initialization: Sets-up and activates the MouseGraphics environment;
Cursor Manager: Sets cursor parameters (e.g., character, visibility);
Event Manager: Reads the mouse and track cursor events;
Menu Manager: Initializes, enables, disables, and reads menus;
Window Manager: Initializes, opens, and closes windows; and,
Control Manager: Sets control parameters for windows.

The **MouseGraphics Toolkit** also contains sample Pascal and Assembly language programs which demonstrate the Toolkit. Finally, the Toolkit includes memory relocation utilities which allow you to bank switch graphics and program modules between the main and alternate 64K banks of memory.

The **MouseGraphics Toolkit** is essential to all Pascal programmers who are now writing, or who plan to write, state-of-the-art software for the Apple II.

Hardware Requirements: //c or //e with 128K of memory
Software Requirements: Kyan Pascal (version 2.0)
Suggested Retail Price: $69.95

Toolkit VI
Code Optimizer

The Code Optimizer Toolkit is designed for the advanced programmer who needs to reduce the code size of an application program and/or increase its runtime speed. The optimizer performs two major functions

1. Modifies the intermediate macro file generated by the compiler so that the Assembler generates "Program Specific Code" (i.e., code which includes only those Runtime Library routines which are specifically required by the program).

2. Replaces certain combinations of compiler-generated macros with optimizer macros which shorten code size and increase the runtime speed of the application program. (The areas most improved by the Optimizer are global variable accesses and record field calculations.)

The Optimizer can reduce the size of a program by as much as fifty percent and, in some cases, almost the double execution speed. The following results were achieved using the Code Optimizer on the Sieve of Erotosthenes.

Program	Compiled Only	Optimized	Improvement
Code Size	12.9K Bytes	3.1K Bytes	9.8K
Runtime Speed	15 seconds	5 seconds	10 seconds

The Source Library also offers the programmer the following advantages:

1. The separate Kyan Pascal Runtime Library is not longer required on the disk. Now the application and the Pascal Runtime Library routine are combined in a single executable file.

2. The programmer can customize library routines (written in assembly langauge source code) to optimize performance and/or meet the specific needs of the application program.

The Code Optimizer Toolkit is a valuable tool for those programmers who are writing large applications and/or who are using MouseText, MouseGraphics, or Advanced Graphics routines in their programs.

Hardware Requirements: Any Apple // with 64K
Software Requirements: Kyan Pascal (version 2.0)
Suggested Retail Price: $149.95

Other Kyan Products

Kyan Binder

Order an extra binder to organize and store your Programming Toolkits. The attractive three ring binder is designed to match the text and divider tab which accompanies each Toolkit.

Kyan Binder ..... $6.95 each/$9.95 outside North America
(Shipping and handling charges do not apply)

COMPUTER GRAPHICS: A Programming Approach

by Steven Harrington (McGraw-Hill Publishing, 1983).

This 445 page textbook is quickly becoming the standard introduction to interactive computer graphics. It is the perfect accompaniment to Kyan's Advanced Graphics Toolkit. It provides the hands-on experience and basic information needed to implement, modify, and use a computer graphics system. The book is built around detailed language independent algorithms for a graphics system and follows the standards proposed in the Graphics Standards Planning Committee's CORE system. By using this standard of basic graphics capabilities, the book provides a solid foundation for more advanced techniques. It includes coverage of raster graphics and discusses interactive techniques, enabling the reader to learn methods of graphical input as well as output. In addition, general 3D viewing is treated to familiarize the reader with the CORE system approach to viewing three-dimensional objects. Numerous problems and experiential exercises are included to enhance comprehension of the material.

COMPUTER GRAPHICS $36.95

Kyan Pascal UpGrades

Kyan Pascal UpGrades

Registered owners of Kyan Pascal (Version 1.+) can upgrade to Kyan Pascal (Version 2.0) or to Kyan Pascal PLUS by returning their original Pascal source disk and payment to Kyan Software. The upgrade charges are:

UpGrade to Kyan Pascal (Version 2.0) $35.00

UpGrade to Kyan Pascal PLUS $50.00

(plus $4.50 for shipping and handling)

The upgrade price includes a completely new manual, source disk(s), and, in the case of Kyan Pascal PLUS, a Quick Reference Guide to KIX and Kyan Pascal.

Update ... Kyan
Kyan's BiMonthly Newsletter

Update ... Kyan is Kyan Software's bimonthly newsletter for users of Kyan Pascal and other programming products. It is intended to provide subscribers with technical information, programming tips, bug reports, upgrade information, and new product announcements. It also contains a number of on-going columns on Pascal and assembly language programming which feature articles of interest for both beginning and advanced programmers.

Through **Update ... Kyan**, we encourage the submission of letters, suggestions, and software routines from users of Kyan products. We also sponsor programming contests, awarding cash payments and gift certificates to those submitting the best programs. The winning listings are then published in the newsletter and/or made available to subscribers on disk for a small handling charge.

Update ... Kyan is our way of keeping in touch with our users and making sure that you are working with the latest in Kyan Software technology. Our readers find that the newsletter is a valuable addition to their technical libraries.

The subscription price is $9.00 per year (6 issues). The discounts offered on new software products usually offsets this charge within the first couple of months. If you write programs using Kyan Software products, you should not go another day without **Update ... Kyan.**

Update ... Kyan: $9.00 per year (6 issues)

(Shipping and handling charges do not apply)